Black Widow: Blackbox Data-driven Web Scanning

Benjamin Eriksson[*], Giancarlo Pellegrino[†], and Andrei Sabelfeld[*]

[*]Chalmers University of Technology
[†]CISPA Helmholtz Center for Information Security

***Abstract*—Modern web applications are an integral part of our digital lives. As we put more trust in web applications, the need for security increases. At the same time, detecting vulnerabilities in web applications has become increasingly hard, due to the complexity, dynamism, and reliance on third-party components. Blackbox vulnerability scanning is especially challenging because (i) for deep penetration of web applications scanners need to exercise such browsing behavior as user interaction and asynchrony, and (ii) for detection of nontrivial injection attacks, such as stored cross-site scripting (XSS), scanners need to discover inter-page data dependencies.**

This paper illuminates key challenges for crawling and scanning the modern web. Based on these challenges we identify three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. While prior efforts are largely limited to the separate pillars, we suggest an approach that leverages all three. We develop Black Widow, a blackbox data-driven approach to web crawling and scanning. We demonstrate the effectiveness of the crawling by code coverage improvements ranging from 63% to 280% compared to other crawlers across all applications. Further, we demonstrate the effectiveness of the web vulnerability scanning by featuring no false positives and finding more cross-site scripting vulnerabilities than previous methods. In older applications, used in previous research, we find vulnerabilities that the other methods miss. We also find new vulnerabilities in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

I. Introduction

Ensuring the security of web applications is of paramount importance for our modern society. The dynamic nature of web applications, together with a plethora of different languages and frameworks, makes it particularly challenging for existing approaches to provide sufficient coverage of the existing threats. Even the web's main players, Google and Facebook, are prone to vulnerabilities, regularly discovered by security researchers. In 2019 alone, Google's bug bounty paid $6.5 million [16] and Facebook $2.2 million [12], both continuing the ever-increasing trend. *Cross-Site Scripting (XSS)* attacks, injecting malicious scripts in vulnerable web pages, represent the lion's share of web insecurities. Despite mitigations by the current security practices, XSS remains a prevalent class of attacks on the web [38]. Google rewards millions of dollars for XSS vulnerability reports yearly [21], and XSS is presently the most rewarded bug on both HackerOne [20] and Bugcrowd [5]. This motivates the focus of this paper on detecting vulnerabilities in web applications, with particular emphasis on XSS.

Blackbox web scanning: When such artifacts as the source code, models describing the application behaviors, and code annotations are available, the tester can use whitebox techniques that look for vulnerable code patterns in the code or vulnerable behaviors in the models. Unfortunately, these artifacts are often unavailable in practice, rendering whitebox approaches ineffective in such cases.

The focus of this work is on *blackbox* vulnerability detection. In contrast to whitebox approaches, blackbox detection techniques rely on no prior knowledge about the behaviors of web applications. This is the standard for security penetration testing, which is a common method for finding security vulnerabilities [31]. Instead, they acquire such knowledge by interacting with running instances of web applications with *crawlers*. Crawlers are a crucial component of blackbox scanners that explore the attack surface of web applications by visiting webpages to discover URLs, HTML form fields, and other input fields. If a crawler fails to cover the attack surface sufficiently, then vulnerabilities may remain undetected, leaving web applications exposed to attacks.

Unfortunately, having crawlers able to discover in-depth behaviors of web applications is not sufficient to detect vulnerabilities. The detection of vulnerabilities often requires the generation of tests that can interact with the web application in non-trivial ways. For example, the detection of stored cross-site scripting vulnerabilities (stored XSS), a notoriously hard class of vulnerabilities [38], requires the ability to reason about the subtle dependencies between the control and data flows of web application to identify the page with input fields to inject the malicious XSS payload, and then the page that will reflect the injected payload.

Challenges: Over the past decade, the research community has proposed different approaches to increase the coverage of the attack surface of web applications. As JavaScript has rendered webpages dynamic and more complex, new ideas were proposed to incorporate these dynamic behaviors to ensure a correct exploration of the page behaviors (jÄk [30]) and the asynchronous HTTP requests (CrawlJAX [26, 4]). Similarly, other approaches proposed to tackle the complexity of the server-side program by reverse engineering (LigRE [10] and KameleonFuzz [11]) or inferring the state (Enemy of the State [8]) of the server, and then using the learned model to drive a crawler.

Unfortunately, despite the recent efforts, existing approaches do not offer sufficient coverage of the attack surface. To tackle this challenge, we start from two observations. First, while prior work provided solutions to individual challenges, leveraging their carefully designed combination has the potential to significantly improve the state of the art of modern web application scanning. Second, existing solutions focus mostly on handling control flows of web applications, falling short of taking into account intertwined dependencies between control and data flows. Consider, for example, the dependency between a page to add new users and the page to show existing users, where the former changes the state of the latter. Being able to extract and use such an inter-page dependency will allow scanners to explore new behaviors and detect more sophisticated XSS vulnerabilities.

Contributions: This paper presents Black Widow, a novel blackbox web application scanning technique that identifies and builds on three pillars: navigation modeling, traversing, and tracking inter-state dependencies.

Given a URL, our scanner creates a navigation model of the web application with a novel JavaScript dynamic analysis-based crawler able to explore both the static structure of webpages, i.e., anchors, forms, and frames, as well as discover and fire JavaScript events such as mouse clicks. Also, our scanner further annotates the model to capture the sequence of steps required to reach a given page, enabling the crawler to retrace its steps. When visiting a webpage, our scanner enriches our model with data flow information using a black-box, end-to-end, dynamic taint tracking technique. Here, our scanner identifies input fields, i.e., taint source, and then probe them with unique strings, i.e., taint values. Later, the scanner checks when the strings re-surface in the HTML document, i.e., sinks. Tracking these taints allows us to understand the dependencies between different pages.

We implement our approach as a scanner on top of a modern browser with a state-of-the-art JavaScript engine. To empirically evaluate it, both in terms of coverage and vulnerability detection, we test it on two sets of web applications and compare the results with other scanners. The first set of web applications are older well-known applications that have been used for vulnerability testing before, e.g. WackoPicko and SCARF. The second set contains new production applications such as CMS platforms including WordPress and E-commerce platforms including PrestaShop and osCommerce. From this, we see that our approach improves code coverage by between 63% and 280% compared to other scanners across all applications. Across all web applications, our approach improves code coverage by between 6% and 62%, compared to the *sum* of all other scanners. In addition, our approach finds more XSS vulnerabilities in older applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, that have been used in previous research. Finally, we also find multiple new vulnerabilities across production software including HotCRP, osCommerce, PrestaShop and WordPress.

Finally, while most scanners produce false positives, Black Widow is free of false positives on the tested applications thanks to its dynamic verification of code injections.

In summary, the paper offers the following contributions.

- We identify unsolved challenges for scanners in modern web applications and present them in Section II.
- We present our novel approaches for finding XSS vulnerabilities using inter-state dependency analysis and crawling complex workflows in Section III.
- We implement and share the source code of Black Widow[1]
- We perform a comparative evaluation of Black Widow on 10 popular web applications against 7 web application scanners.
- We present our evaluation in Section IV showing that our approach finds 25 vulnerabilities, of which 6 are previously unknown in HotCRP, osCommerce, PrestaShop and WordPress. Additionally, we find more vulnerabilities in older applications compared to other scanners. We also improve code coverage on average by 23%.
- We analyze the results and explain the important features required by web scanners in Section V.

II. Challenges

Existing web application scanners suffer from a number of shortcomings affecting their ability to cope with the complexity of modern web applications [9, 3]. We observe that state-of-the-art scanners tend to focus on separate challenges to improve their effectiveness. For example, jÄk focuses on JavaScript events, Enemy of the State on application states, LigRE on reverse engineering and CrawlJAX on network requests. However, to successfully scan applications our insight is that these challenges must be solved simultaneously. This section focuses on these shortcomings and extracts the key challenges to achieve high code coverage and effective vulnerability detection.

High code coverage is crucial for finding any type of vulnerability as the scanner must be able to reach the code to test it. For vulnerability detection, we focus on stored XSS as it is known to be difficult to detect and a category of vulnerabilities poorly covered by existing scanners [9, 3]. Here the server stores and uses at a later time untrusted inputs in server operations, without doing proper validation of the inputs or sanitization of output.

A web application scanner tasked with the detection of subtle vulnerabilities like stored XSS faces three major challenges. First, the scanner needs to model the various states forming a web application, the connections and dependencies between states (Section II-A). Second, the identification of these dependencies requires the scanner to be able to traverse the complex workflows in applications (Section II-B). Finally, the scanner needs to track subtle dependencies between states of the web application (Section II-C).

A. Navigation Modeling

Modern web applications are dynamic applications with an abundance of JavaScript code, client-side events and server-side statefulness. Modeling the scanner's interaction with both server-side and client-side code is complicated and challenging. Network requests can change the state of the server while clicking a button can result in changes to the DOM, which in turn generates new links or fields. These orthogonal problems

[1] Our implementation is available online on https://www.cse.chalmers.se/research/group/security/black-widow/

must all be handled by the scanner to achieve high coverage and improved detection rate of vulnerabilities. Consider the flow in an example web application in Figure 1. The scanner must be able to model links, forms, events and the interaction between them. Additionally, to enable workflow traversal, it must also model the path taken through the application. Finally, the model must support inter-state dependencies as shown by the dashed line in the figure.

The state-of-the-art consists of different approaches to navigation modeling. Enemy of the State uses a state machine and a directed graph to infer the server-side state. However, the navigation model lacks information about client-side events. In contrast, jÄk used a graph with lists inside nodes, to represent JavaScript events. CrawlJAX moved the focus to model JavaScript network requests. While these two model client-side, they miss other important navigation methods such as form submissions.

A navigation model should allow the scanner to efficiently and exhaustively scan a web application. Without correct modeling, the scanner will miss important resources or spend too much time revisiting the same or similar resources. To achieve this, the model must cover a multitude of methods for interaction with the application, including GET and POST requests, JavaScript events, HTML form and iframes.

In addition, the model should be able to accommodate dependencies. Client-side navigations, such as clicking a button, might depend on previous events. For example, the user might have to hover the menu before being able to click the button. Similarly, installation wizards can require a set of forms to be submitted in sequence.

With a solution to the modeling challenge, the next challenge is how the scanner should use this model, i.e. how should it traverse the model.

B. Traversing

To improve code coverage and vulnerability detection, the crawler component of the scanner must be able to traverse the application. In particular, the challenge of reproducing workflows is crucial for both coverage and vulnerability detection. The challenges of handling complex workflows include deciding in which order actions should be performed and when to perform possibly state-changing actions, e.g. submitting forms. Also, the workflows must be modeled at a higher level than network requests as simply replaying requests can result in incorrect parameter values, especially for context-dependent value such as a comment ID. In Figure 1, we can observe a workflow requiring a combination of normal link navigation, form submission and event interaction. Also, note that the forms can contain security nonces to protect against CSRF attacks. A side effect of this is that the scanner can not replay the request and just change the payload, but has to reload the page and resubmit the form.

The current state-of-the-art focuses largely on navigation and exploration but misses out on global workflows. Both CrawlJAX and jÄk focused on exploring client-side events. By exploring the events in a depth-first fashion, jÄk can find sequences of events that could be exploited. However, these sequences do not extend across multiple pages, which will miss out on flows. Enemy of the State takes the opposite approach and ignores traversing client-side events and instead focuses on traversing server-side states. To traverse, they use a combination of picking links from the previous response and a heuristic method to traverse edges that are the least likely to result in a state change, e.g. by avoiding form submission until necessary. To change state they sometimes need to replay the request from the start. Replaying requests may not be sufficient as a form used to post comments might contain a submission ID or view-state information that changes for each request. Due to the challenge of reproducing these flows, their approach assumes the power to reset the full application when needed, preventing the approach from being used on live applications.

We note that no scanner handles combinations of events and classic page navigations. Both jÄk and CrawlJAX traverse with a focus on client-side state while Enemy of the State focus on links and forms for interaction. Simply combining the two approaches of jÄk and Enemy of the State is not trivial as their approaches are tailored to their goals. Enemy of the State uses links on pages to determine state changes, which are not necessarily generated by events.

Keeping the scanner authenticated is also a challenge. Some scanners require user-supplied patterns to detect authentication [34, 28, 36]. jÄk authenticates once and then assumes the state is kept, while CrawlJAX ignores it altogether. Enemy of the State can re-authenticate if they correctly detect the state change when logging out. Once again it is hard to find consensus on how to handle authentication.

In addition to coverage, traversing is important for the fuzzing part of the scanner. Simply exporting all requests to a standalone fuzzer is problematic as it results in loss of context. As such, the scanner must place the application in an appropriate state before fuzzing. Here some scanners take the rather extreme approach of trying to reset the entire web application before fuzzing each parameter [8, 10, 11]. jÄk creates a special attacker module that loads a URL and then executes the necessary events. This shows that in order to fuzz the application in a correct setting, without requiring a full restart of the application, the scanner must be able to traverse and attack both server-side and client-side components.

Solving both modeling and traversing should enable the scanner to crawl the application with improved coverage, allowing it to find more parameters to test. The final challenge, particularly with respect to stored XSS, is mapping the dependencies between different states in the application.

C. Inter-state Dependencies

It is evident that agreeing on a model that fits both client-side and server-side is hard, yet important. In addition, neither of the previous approaches are capable of modeling inter-state dependencies or general workflows. While Enemy of the State model states, they miss the complex workflows and the inter-state dependencies. The model jÄk uses can detect workflows on pages but fails to scale for the full application.

A key challenge faced by scanners is how to accurately and precisely model how user inputs affect web applications.

As an example, consider the web application workflow in Figure 1 capturing an administrator registering a new user. In this workflow, the administrator starts from the index page (i.e., `index.php`) and navigates to the login page (i.e., `login.php`). Then, the administrator submits the password and lands on the administrator dashboard (i.e., `admin.php`). From the dashboard, the administrator reaches the user management page (i.e., `admin.php#users`), and submits the form to register a new user. Then, the web application stores the new user data in the database, and, as a result of that, the data of the new user is shown when visiting the page of existing users (i.e., `view_users.php`). Such a workflow shows two intricate dependencies between two states of the web application: First, an action of `admin.php#users` can cause a transition of `view_users.php`, and second, the form data submitted to `admin.php#users` is reflected in the new state of `admin.php#users`.

To detect if the input fields of the form data are vulnerable to, e.g., cross-site scripting (XSS), a scanner needs to inject payloads in the form of `admin.php#users` and then reach `view_users.php` to verify whether the injection was successful. Unfortunately, existing web scanners are not aware of these inter-state dependencies, and after injecting payloads, they can hardly identify the page where and whether the injection is reflected.

III. Approach

Motivated by the challenges in Section II, this section presents our approach to web application scanning. The three key ingredients of our approach are edge-driven navigation with path-augmentation, complex workflow traversal, and fine-grained inter-state dependency tracking. We explain how we connect these three parts in Algorithm 1. In addition to the three main pillars, we also include a section about the dynamic XSS detection used in Black Widow and motivate why false positives are improbable.

Algorithm 1 takes a single target URL as an input. We start by creating an empty node, allowing us to create an initial edge between the empty node and the node containing the input URL. The main loop picks an unvisited edge from the navigation graph and then traverses it, executing the necessary workflows as shown in Algorithm 2. In Algorithm 2, we use the fact that each edge knows the previous edge. The `isSafe` function in Algorithm 2 checks if the type of action, e.g. JavaScript event or form submission, is *safe*. We consider a type to be *safe* if it is a GET request, more about this in Section III-B. Once the safe edge is found we navigate the chain of actions. Following this navigation, the scanner is ready to parse the page. First, we inspect the page for inter-state dependency tokens and add the necessary dependency edges, as shown in Algorithm 3. Each `token` will contain a taint value, explained more in Section III-C, a source edge and a sink edge. If a source and sink are found, our scanner will fuzz the source and check the sink. Afterward, we extract any new possible navigation resources and add them to the graph. Next, we fuzz any possible parameters in the edge and then inject a taint token. The order is important as we want the token to overwrite any stored fuzzing value. Finally, the edge is marked as visited and the loop repeats.

The goal of this combination is to improve both vulnerability detection and code coverage. The three parts of the approach support each other to achieve this. A strong model that handles different navigation methods and supports augmentation with path and dependency information will enable a richer interaction with the application. Based on the model we can build a strong crawler component that can handle complex workflow which combines requests and client-side events. Finally, by tracking inter-state dependencies we can improve detection of stored vulnerabilities.

```
Data: Target url
Global: tokens // Used in Algorithm 3
Graph navigation; // Augmented navigation graph
navigation.addNode(empty);
navigation.addNode(url);
navigation.addEdge(empty, url);
while unvisited edge e in navigation do
    traverse(e); // See Algorithm 2
    inspectTokens(e, navigation); // See Algorithm 3
    resources = extract({urls, forms, events, iframes});
    for resource in resources do
        navigation.addNode(resource)
        navigation.addEdge(e.targetNode, resource)
    end
    attack(e);
    injectTokens(e);
    mark e as visited;
end
```

Algorithm 1: Scanner algorithm

```
Function traverse(e: edge)
    workflow = []; // List of edges
    currentEdge = e;
    while prevEdge = currentEdge.previous do
        workflow.prepend(currentEdge);
        if isSafe(currentEdge.type) then
            break;
        end
        currentEdge = prevEdge
    end
    navigate(workflow);
end
```

Algorithm 2: Traversal algorithm

```
Function inspectTokens(e: edge, g: graph)
    for token in tokens do
        if pageSource(e) contains token.value then
            token.sink = e;
            g.dependency(token.source, token.sink);
            attack(token.source, token.sink);
        end
    end
end
Function injectTokens(e: edge)
    for parameter in e do
        token.value = generateToken();
        token.source = e;
        tokens.append(token);
        inject token in parameter;
    end
end
```

Algorithm 3: Inter-state dependency algorithms

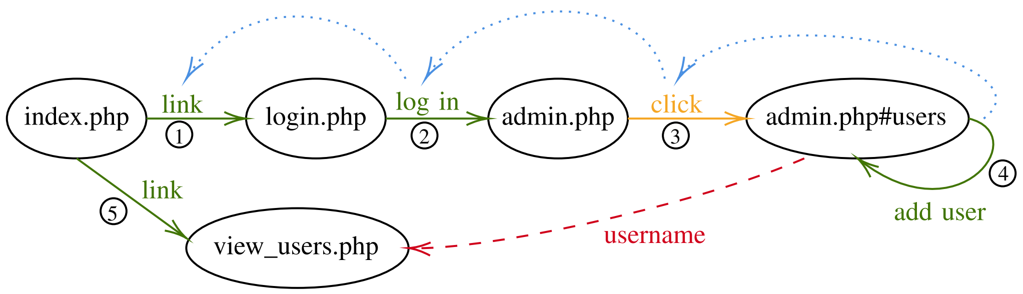

Fig. 1: Example of a web application where anyone can see the list of users and the admin can add new users. The dashed red line represents the inter-state dependency. Green lines are HTML5 and orange symbolises JavaScript. The dotted blue lines between edges would be added by our scanner to track its path. The sequence numbers shown the necessary order to find the inter-state dependency.

A. *Navigation Modeling*

Our approach is model-based in the sense that it creates, maintains, and uses a model of the web application to drive the exploration and detection of vulnerabilities. Our model covers both server-side and client-side aspects of the application. The model tracks server-side inter-state dependencies and workflows. In addition, it directly captures elements of the client-side program of the web application, i.e., HTML and the state of the JavaScript program.

a) Model Construction: Our model is created and updated at run-time while scanning the web application. Starting from an initial URL, our scanner retrieves the first webpage and the referenced resources. While executing the loaded JavaScript, it extracts the registered JavaScript events and adds them to our model. Firing an event may result in changing the internal state of the JavaScript program, or retrieving a new page. Our model captures all these aspects and it keeps track of the sequence of fired events when revisiting the web application, e.g., for the detection of vulnerabilities.

Accordingly, we represent web applications with a labeled directed graph, where each node is a state of the client-side program and edges are the action (e.g., click) to move from one state to another one. The state of our model contains both the state of the page, i.e., the URL of the page, and the state of the JavaScript program, i.e., the JavaScript event that triggered the execution. Then, we use labeled edges for state transitions. Our model supports four types of actions, i.e., GET requests, form submission, iframes and JavaScript events. While form submissions normally result in GET or POST requests, we need a higher-level model for the traversing method explained in Section III-B. We consider iframes as actions because we need to model the inter-document communication between the iframe and the parent, e.g firing an event in the parent might affect the iframe. By simply considering the iframe source as a separate URL, scanners will miss this interaction. Finally, we annotate each edge with the previous edge visited when crawling the web application, as shown in Figure 1. Such an annotation will allow the crawler to reconstruct the *paths* within the web application, useful information for achieving deeper crawling and when visiting the web application for testing.

b) Extraction of Actions: The correct creation of the model requires the ability to extract the set of possible actions from a web page. Our approach uses dynamic analysis approach, where we load a page and execute it in a modified browser environment, and then we observe the execution of the page, monitoring for calls to browser APIs to register JavaScript events and modification of the DOM tree to insert tags such as forms and anchors.

Event Registration Hooking Before loading a page we inject JavaScript which allows us to wrap functions such as `addEventListener` and detect DOM objects with event handlers. We accomplish this by leveraging the JavaScript libraries developed for the jÄk scanner [30]. While lightweight and easy to use, in-browser instrumentation is relatively fragile. A more robust approach could be directly modifying the JavaScript engine or source-to-source compile the code for better analysis.

DOM Modification To detect updates to the page we rescan the page whenever we execute an event. This allows us to detect dynamically added items.

c) Infinite Crawls: When visiting a webpage, crawlers can enter in an infinite loop where they can perform the same operation endlessly. Consider the problem of crawling an online calendar. When a crawler clicks on the *View next week* button, the new page may have a different URL and content. The new page will container again the button *View next week*, triggering an infinite loop. An effective strategy to avoid infinite crawls is to define (i) a set of heuristics that determine when two pages or two actions are similar, and (ii) a hard limit to the maximum number of "similar" actions performed by the crawler. In our approach, we define two pages to be similar if they share the same URL except for the query string and the fragments. For example, `https://example.domain/path/?x=1` and `https://example.domain/path/?x=2` are similar whereas `https://example.domain/?x=1` and `https://example.domain/path/?x=2` are different. The hard limit is a configuration parameter of our approach.

B. Traversal

To traverse the navigation model we pick unvisited edges from the graph in the order they were added, akin to breadth-first search. This allows the scanner to gain an overview of the application before diving into specific components. The edges are weighted with a positive bias towards form submission, which enables this type of deep-dive when forms are detected.

To handle the challenge of session management, we pay extra attention to forms containing password fields, as this symbolizes an opportunity to authenticate. Not only does this enable the scanner to re-authenticate but it also helps when the application generates a login form due to incorrect session tokens. Another benefit is a more robust approach to complicated login flows, such as double login to reach the administrator page—we observed such workflow in phpBB, one of the web applications that we evaluated.

The main challenge to overcome is that areas of a web application might require the user to complete a specific sequence of actions. This could, for example, be to review a comment after submitting it or submit a sequence of forms in a configuration wizard. It is also common for client-side code to require chaining, e.g. hover a menu before seeing all the links or click a button to dynamically generate a new form.

We devise a mechanism to handle navigation dependencies by modeling the workflows in the application. Whenever we need to follow an edge in the navigation graph, we first check if the previous edge is considered *safe*. Here we define safe to be an edge which represents a GET request, similar to the HTTP RFC [14]. If the edge is safe, we execute it immediately. Otherwise, we recursively inspect the previous edge until a safe edge is found, as shown in Algorithm 2. Note that the first edge added to the navigation graph is always a GET request, which ensures a base case. Once the safe edge is found, we execute the full workflow of edges leading up to the desired edge. Although the RFC defines GET requests to be idempotent, developers can still implement state-changing functions on GET requests. Therefore, considering GET requests as *safe* is a performance trade-off. This could be deactivated by a parameter in Black Widow, causing the scanner to traverse back to the beginning.

Using Figure 1 as an example if the crawler needed to submit a form on `admin.php#users` then it would first have to load `login.php` and then submit that form, followed by executing a JavaScript event to dynamically add the user form.

We chose to only chain actions to the previous GET request, as they are deemed safe. Chaining from the start is possible, but it would be slow in practice.

C. Inter-state Dependencies

One of the innovative aspects of our approach is to identify and map the ways user inputs are connected to the states of a web application. We achieve that by using a dynamic, end-to-end taint tracking while visiting the web application. Whenever our scanner identifies an input field, i.e., a *source*, it will submit a unique token. After that, the scanner will look for the token when visiting other webpages, i.e., *sinks*.

a) Tokens: To map source and sinks, we use string tokens. We designed tokens to avoid triggering filtering functions or data validation checks. At the same time, we need tokens with a sufficiently high entropy to not be mistaken for other strings in the application. Accordingly, we generate tokens as pseudo-random strings of eight lowercase characters e.g. `frcvwwzm`. This is what `generateToken()` does in Algorithm 3. This could potentially be improved by making the tokens context-sensitive, e.g. by generating numeric tokens or emails. However, if the input is validated to only accept numbers, for example, then XSS is not possible.

b) Sources and Sinks: The point in the application where the token is injected defines the *source*. More specifically, the source is defined as a tuple containing the edge in the navigation graph and the exact parameter where the token was injected. The resource in the web application where the token reappears defines the *sink*. All the sinks matching a certain source will be added to a set which in turn is connected to the source. Similar to the sources, each sink is technically an edge since they carry more context than a resource node. Since each source can be connected to multiple sinks, the scanner needs to check each sink for vulnerabilities whenever a payload is injected into a source.

In our example in Figure 1, we have one source and one connected sink. The source is the *username* parameter in the form on the management page and the sink is the view users page. If more parameters, e.g. email or signature, were also reflected then these would create new dependency edges in the graph.

D. Dynamic XSS detection

After a payload has been sent, the scanner must be able to detect if the payload code is executed. Black Widow uses a fine-grained dynamic detection mechanism, making false positives very improbable. We achieve this by injecting our JavaScript function `xss(ID)` on every page. This function adds `ID` to an array that our scanner can read. Every payload generated by Black Widow will try to call this function with a random ID, e.g. `<script>xss(71942203)</script>` Finally, by inspecting the array we can detect exactly which payloads resulted in code execution.

For this to result in a false positive, the web application would have to actively listen for a payload, extract the ID, and then run our injected `xss(ID)` function with a correct ID.

IV. Evaluation

In this section, we present the evaluation of our approach and the results from our experiments. In the next section, we perform an in-depth analysis of the factors behind the results.

To evaluate the effectiveness of our approach we implement it in our scanner Black Widow and compare it with 7 other scanners on a set of 10 different web applications. We want to compare both the crawling capabilities and vulnerability detection capabilities of the scanners. We present the implementation details in Section IV-A. The details of the experimental setup are presented in Section IV-B. To measure the crawling capabilities of the scanners we record the code coverage on each of application. The code coverage results are presented in Section IV-C. For the vulnerability detection capabilities, we collect the reports from each scanner. We present both the reported vulnerabilities and the manually verified ones in Section IV-D.

A. Implementation

Our prototype implementation follows the approach presented above in Section III. It exercises full dynamic execution capabilities to handle such dynamic features of modern applications like AJAX and dynamic code execution, e.g. `eval`. To achieve this we use Python and Selenium to control a mainstream web browser (Chrome). This gives us access to a state-of-the-art JavaScript engine. In addition, by using a mainstream browser we can be more certain that the web application is rendered as intended.

We leverage the JavaScript libraries developed for the jÄk scanner [30]. These libraries are executed before loading the page. This allows us to wrap functions such as `addEventListener` and detect DOM objects with event handlers.

B. Experimental Setup

In this section, we present the configuration and methodology of our experiments.

a) Code Coverage: To evaluate the coverage of the scanners we chose to compare the lines of code that were executed on the server during the session. This is different from previous studies [8, 30], which relied on requested URLs to determine coverage. While comparing URLs is easier, as it does not require the web server to run in debug mode, deriving coverage from it becomes harder. URLs can contain random parameter data, like CSRF tokens, that are updated throughout the scan. In this case, the parameter data has a low impact on the true coverage. Conversely, the difference in coverage between `main.php?page=news` and `main.php?page=login` can be large. By focusing on the execution of lines of code we get a more precise understanding of the coverage.

Calculating the total number of lines of code accurately in an application is a difficult task. This is especially the case in languages like PHP where code can be dynamically generated server-side. Even if possible, it would not give a good measure for comparison as much of the code could be unreachable. This is typically the case for applications that have installation code, which is not used after completing it.

Instead of analyzing the fraction of code executed in the web application, we compare the number of lines of code executed by the scanners. This gives a relative measure of performance between the scanners. It also allows us to determine exactly which lines are found by multiple scanners and which lines are uniquely executed.

To evaluate the code coverage we used the Xdebug [33] module in PHP. This module returns detailed data on the lines of code that are executed in the application. Each request to the application results in a separate list of lines of code executed for the specific request.

b) Vulnerabilities: In addition to code coverage, we also evaluate how good the scanners are at finding vulnerabilities. This includes how many vulnerabilities they can find and how many false positives they generate. While there are many vulnerability types, our study focuses on both reflected and stored XSS.

To evaluate the vulnerability detection capabilities of the scanners, we collect and process all the vulnerabilities they report. First, we manually analyze if the vulnerabilities can be reproduced or if they should be considered false positives. Second, we cluster similar vulnerability reports into a set of unique vulnerabilities to make a fair comparison between the different reporting mechanisms in the scanners. We do this because some applications, e.g. SCARF, can generate an infinite number of vulnerabilities by dynamically adding new input fields. These should be clustered together. Classifying the uniqueness of vulnerabilities is no easy task. What we aim to achieve is a clustering in which each injection corresponds to a unique line of code on the server. That is, if a form has multiple fields that are all stored using the same SQL query then all these should count as one injection. The rationale is that it would only require the developer to change one line in the server code. Similarly, for reflected injections, we cluster parameters of the same request together. We manually inspect the web application source code for each reported true-positive vulnerability to determine if they should be clustered.

c) Scanners: We compare our scanner Black Widow with both Wget [27] for code coverage reference and 6 state-of-the-art open-source web vulnerability scanners from both academia and the web security community: Arachni [36], Enemy of the State [8], jÄk [30], Skipfish [42], w3af [34] and ZAP [28]. We use Enemy of the State and jÄk as they are state-of-the-art academic blackbox scanners. Skipfish, Wget and w3af are included as they serve as good benchmarks when comparing with previous studies [8, 30]. Arachni and ZAP are both modern open-source scanners that have been used in more recent studies [19]. Including a pure crawler with JavaScript capabilities, such as CrawlJAX [26], could serve as a good coverage reference. However, in this paper we focus on coverage compared to other vulnerability scanners. We still include Wget for comparison with previous studies. While it would be interesting to compare our results with commercial scanners, e.g. Burp Scanner [32], the closed source nature of these tools would make any type of feature attribute hard.

We configure the scanners with the correct credentials for the web application. When this is not possible we change the default credentials of the application to match the scanner's default values. Since the scanners have different capabilities, we try to configure them with as similar configurations as possible. This entails activating crawling components, both static and dynamic, and all detection of all types of XSS vulnerabilities.

Comparing the time performance between scanners is non-trivial to do fairly as they are written in different languages and some are sequential while others run in parallel. Also, we need to run some older ones in VMs for compatibility reasons. To avoid infinite scans, we limit each scanner to run for a maximum of eight hours.

TABLE I: Lines of code (LoC) executed on the server. Each column represents the comparison between Black Widow and another crawler. The cells contain three numbers: unique LoC covered by Black Widow ($A \setminus B$), LoC covered by both crawlers ($A \cap B$) and unique LoC covered by the other crawler ($B \setminus A$). The numbers in bold highlight which crawler has the best coverage.

Crawler	Arachni			Enemy			jÄk			Skipfish			w3af			Wget			ZAP		
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$
Drupal	**35 146**	22 870	757	6 365	51 651	**20 519**	**25 198**	32 818	5 846	**29 873**	28 143	937	**32 213**	25 803	725	**32 981**	25 035	498	**15 610**	42 406	2 591
HotCRP	**2 416**	16 076	948	**16 573**	1 919	0	**6 771**	11 721	271	**11 295**	7 197	31	**3 217**	15 275	768	**16 345**	2 147	3	**16 001**	2 491	24
Joomla	**14 573**	29 263	1 390	**33 335**	10 501	621	**24 728**	19 108	1 079	**33 254**	10 582	328	**12 533**	31 303	1 255	**33 975**	9 861	576	**7 655**	36 181	1 659
osCommerce	**3 919**	6 722	172	**9 626**	1 015	15	**4 171**	6 470	507	**4 964**	5 677	110	**5 601**	5 040	661	**6 070**	4 571	103	**6 722**	3 919	209
phpBB	**2 822**	5 178	492	**2 963**	5 037	337	**3 150**	4 850	348	**4 643**	3 357	72	**4 312**	3 688	79	**4 431**	3 569	21	**4 247**	3 753	65
PrestaShop	**105 974**	75 924	65 650	**157 095**	24 803	3 332	**155 579**	26 319	58	**138 732**	43 166	1 018	**156 513**	25 385	3 053	**148 868**	33 030	118	**141 032**	40 866	110
SCARF	**189**	433	12	**270**	352	5	**342**	280	2	**464**	158	5	**404**	218	6	**520**	102	2	**340**	282	2
Vanilla	**5 381**	9 908	491	**6 032**	9 257	185	**3 122**	12 167	536	**8 285**	7 004	577	**8 202**	7 087	171	**8 976**	6 313	18	**8 396**	6 893	145
WackoPicko	**202**	566	2	**58**	710	9	**463**	305	0	**274**	494	14	**111**	657	9	**495**	273	0	**379**	389	2
WordPress	**8 871**	45 345	1 615	**35 092**	19 124	256	**18 572**	35 644	579	**7 307**	46 909	5 114	**26 785**	27 431	640	**37 073**	17 143	73	**25 732**	28 484	781

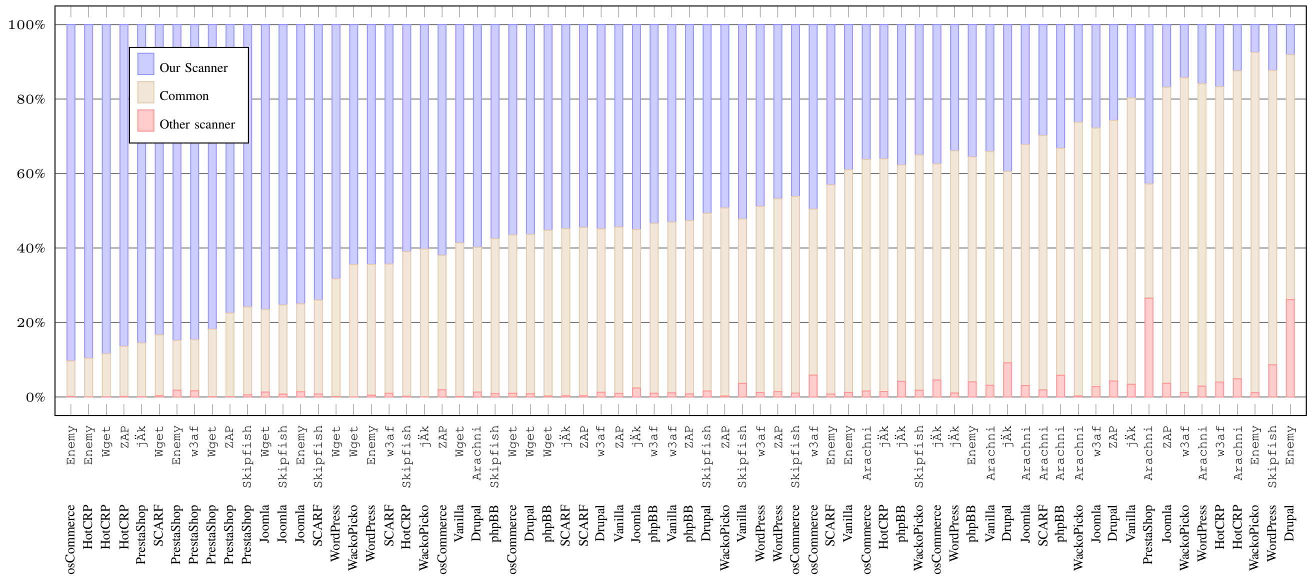

Fig. 2: Each bar compares our scanner to one other scanner on a web application. The bars show three fractions: unique lines we find, lines both find and lines uniquely found by the other scanner.

d) Web Applications: To ensure that the scanners can handle different types of web applications we test them on 10 different applications. The applications range from reference applications that have been used in previous studies to newer production-grade applications. Each application runs in a VM that we can reset between runs to improve consistency.

We divide the applications into two different sets. Reference applications with known vulnerabilities: phpBB (2.0.23), SCARF (2007), Vanilla (2.0.17.10) and WackoPicko (2018); and modern production-grade applications: Drupal (8.6.15), HotCRP (2.102), Joomla (3.9.6), osCommerce (2.3.4.1), PrestaShop (1.7.5.1) and WordPress (5.1).

C. Code Coverage Results

This section presents the code coverage in each web application by all of the crawlers. Table I shows the number of unique lines of code that were executed on the server. Black Widow has the highest coverage on 9 out of the 10 web applications.

Using Wget as a baseline Table I illustrates that Black Widow increases the coverage by almost 500% in SCARF. Similarly with modern production software, like PrestaShop, we can see an increase of 256% in coverage compared to Wget. Even when comparing to state-of-the-art crawlers like jÄk and Enemy of the State we have more than 100% increase on SCARF and 320% on modern applications like PrestaShop. There is, however, a case where Enemy of the State has the highest coverage on Drupal. This case is discussed in more detail in Section V-A.

While it would be beneficial to know how far we are from perfect coverage, we avoid calculating a ground truth on the total number of lines of code for the applications as it is difficult to do in a meaningful way. Simply aggregating the number of lines in the source code will misrepresent dynamic code, e.g. `eval`, and count dead code, e.g. installation code.

We also compare Black Widow to the combined efforts of the other scanners to better understand how we improve the state-of-the-art. Table II has three columns containing the number of lines of code that Black Widow finds which none of the others find, the combined coverage of the others and finally our improvement in coverage. In large applications, like

TABLE II: Unique lines our scanner finds ($A \setminus U$) compared to the union of all other scanners (U).

Application	Our scanner $A \setminus U$	Other scanners U	Improvement $\lvert A \setminus U\rvert / \lvert U\rvert$
Drupal	4 378	80 213	+5.5%
HotCRP	1 597	18 326	+8.7%
Joomla	5 134	42 443	+12.1%
osCommerce	2 624	9 216	+28.5%
phpBB	2 743	5 877	+46.7%
PrestaShop	95 139	153 452	+62.0%
SCARF	176	464	+37.9%
Vanilla	2 626	14 234	+18.4%
WackoPicko	50	742	+6.7%
WordPress	3 591	58 131	+6.2%

TABLE IV: Number of reported XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	8	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	9	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	32	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	2	-	-	-
SCARF	31	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	2	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	13	-	1	1	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

TABLE V: Number of unique and correct XSS injections by the scanners and the classification of the injection as either reflected or stored.

Crawler	Arachni		Enemy		jÄk		Skipfish		w3af		Widow		ZAP	
Type	R	S	R	S	R	S	R	S	R	S	R	S	R	S
Drupal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HotCRP	-	-	-	-	-	-	-	-	-	-	1	-	-	-
Joomla	-	-	-	-	-	-	-	-	-	-	-	-	-	-
osCommerce	-	-	-	-	-	-	-	-	-	-	1	1	-	-
phpBB	-	-	-	-	-	-	-	-	-	-	-	3	-	-
PrestaShop	-	-	-	-	-	-	-	-	-	-	1	-	-	-
SCARF	3	-	-	-	-	-	-	-	1	-	3	5	-	-
Vanilla	-	-	-	-	-	-	-	-	-	-	1	2	-	-
WackoPicko	3	1	2	1	1	-	1	-	1	-	3	2	-	-
WordPress	-	-	-	-	-	-	-	-	-	-	1	1	-	-

PrestaShop, Black Widow was able to find 53 266 lines of code that none of the others found. For smaller applications, like phpBB, we see an improvement of up to 46.7% compared to the current state-of-the-art.

To get a better understanding of which parts of the application the scanners are exploring, we further compare the overlap in the lines of code between the scanners. In Table III we present the number of unique lines of code Black Widow find compared to another crawler. The improvement is calculated as the number of unique lines we find divided by the *total* number of lines the other crawlers find.

We plot the comparison for all scanners on all platforms in Figure 2. In this figure, each bar represents the fraction of lines of code attributed to each crawler. At the bottom is the fraction found only by the other crawlers, in the middle the lines found by both and on top are the results found by Black Widow. The bars are sorted by the difference of unique lines found by Black Widow and the other crawlers. Black Widow finds the highest number of unique lines of code in all cases except the rightmost, in which Enemy of the State performed better on Drupal. The exact number can be found in Table I.

D. Code Injection Results

This section presents the results from the vulnerabilities the different scanners find. To be consistent with the terminology used in previous works [8, 30], we define an XSS vulnerability to be any injected JavaScript code that results in execution. While accepting JavaScript from users is risky in general, some applications, like Wordpress, have features which require executing user supplied JavaScript. In Section V-G we discuss the impact and exploitability of the vulnerabilities our scanner finds.

In Table IV we list all the XSS vulnerabilities found by the scanners on all the applications. The table contains the number of self-reported vulnerabilities. After removing the false positives and clustering similar injections, as explained in Section IV-B, we get the new results in Table V. The results from Table V show that Black Widow outperforms the other scanners on both the reference applications and the modern applications. In total, Black Widow finds 25 unique vulnerabilities, which is more than 3 times as many as the second-best scanner. Of these 25, 6 are previously unknown vulnerabilities in modern applications. We consider the remaining 19 vulnerabilities to be known for the following reasons. First, all WackoPicko vulnerabilities are implanted by the authors and they are all known by design. Second, SCARF has been researched thoroughly and vulnerabilities may be already known. We conservatively assumed the eight vulnerabilities to be known. Third, the vulnerabilities on phpBB and Vanilla were fixed in their newest versions.

It is important to note we did not miss any vulnerability that the others found. However, there were cases where both Black Widow and other scanners found the same vulnerability but by injecting different parameters. We explore these cases more in Section V-F. Furthermore, Black Widow is the only scanner that finds vulnerabilities in the modern web applications.

TABLE III: Comparison of unique lines of code found by our scanner ($A \setminus B$) and the other scanners ($B \setminus A$). Improvement is new lines found by our scanner divided by the other's total.

Crawler	Our scanner $A \setminus B$	Other scanners $B \setminus A$	Other's total B	Improvement $\lvert A \setminus B\rvert / \lvert B\rvert$
Arachni	179 477	71 489	283 664	+63.3%
Enemy	267 372	25 268	149 548	+178.8%
jÄk	242 066	9 216	158 802	+152.4%
Skipfish	239 064	8 206	160 794	+148.7%
w3af	249 881	7 328	149 099	+167.6%
Wget	289 698	1 405	103 359	+280.3%
ZAP	226 088	5 560	171 124	+132.1%

E. Takeaways

We have shown that our scanner can outperform the other scanners in terms of both code coverage and vulnerability detection. Figure 2 and Table I show that we outperform the other scanners in 69 out of 70 cases. Additionally, Table II and Table III show we improve code coverage by between 63% and 280% compared to the other scanners and by between 6% and 62%, compared to the *sum* of all other scanners. We also improve vulnerability detection, as can be seen in Table IV and Table V. Not only do we match the other scanners but we also find new vulnerabilities in production applications.

In the next section, we will analyze these results closer and conclude which features allowed us to improve coverage and vulnerability detection. We also discuss what other scanners did better than us.

V. Analysis of Results

The results from the previous section show that the code coverage of our scanner outperforms the other ones. Furthermore, we find more code injections and in particular more stored XSS. In this section, we analyze the factors which led to our advantage. We also analyze where and why the other scanners performed worse.

Since we have access to the executed lines of code we can closely analyze the path of the scanner through the application. We utilize this to analyze when the scanners miss injectable parameters, what values they submit, when they fail to access parts of the application and how they handle sessions.

We start by presenting interesting cases from the code coverage evaluation in Section V-A, followed by an analysis of the reported vulnerabilities from all scanners in Section V-B. In Section V-C, we discuss the injections our scanner finds and compare it with what the others find. In Section V-D, we perform two case studies of vulnerabilities that only our scanner finds and which requires both workflow traversal and dependency analysis. Finally, in Section V-E, we extract the crucial features for finding injections based on all vulnerabilities that were found.

A. Coverage Analysis

As presented in Section IV-C, Black Widow improved code coverage, compared to the aggregated result of all the other scanners, ranged from 5.5% on Drupal to 62% on PrestaShop. Comparing the code coverage to each scanner, Black Widow's improvement ranged from 63.3% against Arachni to 280% against Wget. In this section, we analyze the factors pertaining to code coverage by inspecting the performance of the different scanners. To better understand our performance we divide the analysis into two categories. We look at both cases where we have low coverage compared to the other scanners and cases where we have high relative coverage.

a) Low coverage: As shown in Figure 2, Enemy of the State is the only scanner that outperforms Black Widow and this is specifically on Drupal. Enemy of the State high coverage on Drupal is because it keeps the authenticated session state by avoiding logging out. The reason Black Widow lost the state too early was two-fold. First, we use a heuristic algorithm, as explained in Section III-B to select the next edge and unfortunately the logout edge was picked early. Second, due to the structure of Drupal, our scanner did not manage to re-authenticate. In particular, this is because, in contrast to many other applications, Drupal does not present the user with a login form when they try to perform an unauthorized operation. To isolate the reason for the lower code coverage, we temporarily blacklist the Drupal logout function in our scanner. This resulted in our scanner producing similar coverage to Enemy of the State, ensuring the factor behind the discrepancy is session handling.

Skipfish performs very well on WordPress, which seems surprising since it is a modern application that makes heavy use of JavaScript. However, WordPress degrades gracefully without JavaScript, allowing scanners to find multiple pages without using JavaScript. Focusing on static pages can generate a large coverage but, as is evident from the detected vulnerabilities, does not imply high vulnerability detection.

b) High coverage: Enemy of the State also performs worse against Black Widow on osCommerce and HotCRP. This is because Enemy of the State is seemingly entering an infinite loop, using 100% CPU without generating any requests. This could be due to an implementation error or because the state inference becomes too complicated in these applications.

Although Black Widow performs well against Wget, Wget still finds some unique lines, which can seem surprising as it has previously been used as a reference tool [8, 30]. Based on the traces and source code, we see that most of the unique lines of code Wget finds are due to state differences, e.g. visiting the same page Black Widow finds but while being unauthenticated.

B. False positives and Clustering

To better understand the reason behind the false positives, and be transparent about our clustering, we analyze the vulnerabilities reported in Table IV. For each scanner with false positives, we reflect on the reasons behind the incorrect classification and what improvements are required. We do not include w3af in the list as it did not produce any false positives or required any clustering.

a) Arachni reports two reflected XSS vulnerabilities in Vanilla. The injection point was a Cloudflare cookie used on the online support forum for the Vanilla web application. The cookie is never used in the application and we were unable to reproduce the injection. In addition, Arachni finds 31 XSS injections on SCARF. Many of these are incorrect because Arachni reuses payloads. For example, by injecting into the title of the page, all successive injection will be label as vulnerable.

b) Enemy of the State claims the discovery of 8 reflected XSS vulnerabilities on Joomla. However, after manual analysis, none of these result in code execution. The problem is that Enemy of the State interprets the reflected payload as an executed payload. It injects, `eval(print "[random]")`, into a search field and then detects that `"[random]"` is reflected. It incorrectly assumes this is because `eval` and `print` were executed. For this reason, we consider Enemy of the State to find 0 vulnerabilities on Joomla.

c) jÄk reports 13 vulnerabilities on WackoPicko. These 13 reports were different payloads used to attack the search parameter. After applying our clustering method, we consider jÄk to find one unique vulnerability.

d) Black Widow finds 32 stored vulnerabilities on phpBB. Most of these parameters are from the configuration panel and are all used in the same database query. Therefore, only 3 can be considered unique. Two parameters on PrestaShop are used in the same request, thus only one is considered unique. Black Widow did not produce any false positives thanks to our dynamic detection method explained in section III-D

e) Skipfish claims the detection of a stored XSS in WackoPicko in the image data parameter when uploading an image. However, the injected JavaScript could not be executed. Interesting to note is that Skipfish was able to inject JavaScript into the guestbook but was not able to detect it.

f) ZAP claims to find 9 reflected XSS injection on osCommerce. They are all variations of injecting `javascript:alert(1)` into the parameter of a link. Since it was just part of a parameter and not a full URL, the JavaScript code will never execute. Thus, all 9 injections were false positives.

C. What We Find

In this section, we present the XSS injections our scanner finds in the different applications. We also extract the important features which made it possible to find them.

a) HotCRP: Reflected XSS in bulk user upload: The admin can upload a file with users to add them in bulk. The name of the file is then reflected on the upload page. To find this, the scanner must be able to follow a complex workflow that makes heavy use of JavaScript, as well as handle file parameters. It is worth noting that the filename is escaped on other pages in HotCRP but missed in this case.

b) osCommerce; Stored and reflected XSS: Admins can change the tax classes in osCommerce and two parameters are not correctly filtered, resulting in stored XSS vulnerabilities. The main challenge to find this vulnerability was to find the injection point as this required us to interact with a navigation bar that made heavy use of JavaScript.

We also found three vulnerable parameters on the review page. These parameters were part of a form and their types were *radio* and *hidden*. This highlights that we still inject all parameters, even if they are not intended to be changed.

c) phpBB; Multiple Stored XSS in admin backend: Admins can change multiple different application settings on the configuration page, such as flooding interval for posts and max avatar file size. On a separate page, they can also change the rank of the admin to a custom title. In total, this results in 32 vulnerable parameters that can be clustered to 3 unique ones. These require inter-state dependency analysis to solve. Once a setting is changed, the admin is met with a "Successful update" message, which does not reflect the injection. Thus, the dependency must be found to allow for successful fuzzing.

d) PrestaShop; Reflected XSS in admin dashboard: The admin dashboard allows the admin to specify a date range for showing statistics. Two parameters in this form are not correctly filtered and result in a reflected XSS. Finding these requires a combination of modeling JavaScript events and handling workflows. To find this form the scanner must first click on a button on the dashboard.

e) SCARF; Stored XSS in comments: There are many vulnerabilities in SCARF, most are quite easy to find. Instead of mentioning all, we focus on one that requires complex workflows, inter-state dependencies and was only found by us. The message field in the comment section of conference papers is vulnerable. What makes it hard to find is the traversing and needed before posting the comment and the inter-state dependency analysis needed to find the reflection. The scanner must first create a user, then create a conference, after which it can upload a paper that can be commented on.

f) Vanilla; Stored and reflected XSS: The language tag for the RSS feed is vulnerable and only reflected in the feed. Note that the feed is served as HTML, allowing JavaScript to execute. There is also a stored vulnerability in the comment section which can be executed by saving a comment as a draft and then viewing it. Both of these require inter-state dependency analysis to find the connecting between language settings and RSS feeds, as well as posting comments and viewing drafts.

Black Widow also found a reflected XSS title parameter in the configuration panel that was vulnerable. Finding this mainly required and modeling JavaScript and forms.

g) WackoPicko; Multi-step stored XSS: We found all the known XSS vulnerabilities [7], except the one requiring flash as we consider it out-of-scope. We also found a non-listed XSS vulnerability in the reflection of a SQL error. Most notably we were able to detect the multi-step XSS vulnerability that no other scanner could. This was thanks to both inter-state dependency tracking and handling workflows. We discuss this in more detail in the case study in Section V-D1.

h) WordPress; Stored and reflected XSS: The admin can search for nearby events using the admin dashboard. The problem is that the search query is reflected, through AJAX, for the text-to-speech functionality. Finding this requires modeling of both JavaScript events, network requests and forms.

Our scanner also found that by posting comments from the admin panel JavaScript is allowed to run on posts. For this, the scanner must handle the workflows needed to post the comments and the inter-state dependency analysis needed to later find the comment on a post.

D. Case Studies

In this section, we present two in-depth case studies of vulnerabilities that highlights how and why our approach finds vulnerabilities the other scanners do not. We base our analysis on server-side traces, containing the executed lines of code, generated from the scanner sessions. By manually analyzing the source code of an application we can determine the exact lines of code that need to be executed for an injection to be successful.

The cases we use are the comment section in WackoPicko and the configuration panel in phpBB. As we have shown, Black Widow can find vulnerabilities in more complex modern web applications. Nevertheless, these cases allow us to limit the number of factors when comparing our approach with the other scanners. Since WackoPicko and phpBB have been used in previous studies [30, 8] they also serve as a level playing field for all scanners.

1) Comments on WackoPicko: WackoPicko has a previously unsolved multistep XSS vulnerability that no other scanner has been able to find. The difficultly of finding and exploiting is the need for correctly reproducing a specific workflow. After submitting a comment via a form the user needs to review the comment. While reviewing, the user can choose to either delete the comment or add it. If, however, the user decided to visit another page, before adding or deleting, then the review form will be removed and the user will have to resubmit the comment before reviewing it again. Thus, the steps that must be taken are: Find an image to comment on (`view.php#50`, i.e. line 50 in `view.php`), Post a comment (`preview_comment.php#54`), Accept the comment while reviewing (`view.php#53`) In Table VI we note that two scanners are able to find the input but not exploit it.

Both Enemy of the State and Arachni managed to post a comment but neither could exploit the vulnerability. Enemy of the State was able to post a comment containing an empty string but the fuzzing was unsuccessful. Arguably, Arachni made it a bit further since it was able to inject an XSS payload. However, the payload was not detected and reported. Enemy of the State's shortcoming is that it fuzzes the forms independently while Arachni's shortcoming is that it forgets it's own injection.

jÄk and ZAP had problems finding the first step, i.e. viewing the pictures, because the login form breaks the HTML standard by putting a form inside a table [41]. We avoid this by using a modern browser to parse the web page. This allows Black Widow to view the web page as the developer intended, assuming they tested it in a modern browser

Both w3af and Skipfish were able to find the pictures but not able to post the comment. w3af because it could not model the `textarea` in the form. Skipfish, on the other hand, does not have this problem. We believe that Skipfish logged out after seeing the picture but before posting the comment. The data shows that Skipfish does not try to log in multiple times. In comparison, we correctly handle the `textarea` allowing us to post comments. At the same time, we also try to log in multiple times if presented with a login form. This mitigates losing the session forever at an early stage.

To solve this challenge Black Widow needs to combine the modeling of form elements, handle workflows and use inter-state dependency analysis to correctly inject and detect the vulnerability.

2) Configuration on phpBB: The configuration panel on phpBB has multiple code injection possibilities. To find these the crawler must overcome two challenges. First, to reach the admin panel requires two logins, the first to authenticate as a user and then again, with the same credentials, to authenticate as an administrator. Second, the injected parameter is not reflected on the same page. To detect this injection inter-state dependency analysis is required. The steps needed to find the vulnerability is, log in as admin (`admin/index.php#28`), find the vulnerable form (`admin_board.php#34`), successfully update the database (`admin_board.php#74`) find the reflection (`admin_board.php#34`).

TABLE VI: Steps to recreate the vulnerability in WackoPicko. The columns contain the file name and line of code for each step.

Crawler	view.php#50	preview_comment.php#54	view.php#53	Exploit
Arachni	✓	✓	✓	
Enemy	✓	✓	✓	
jÄk				
Skipfish	✓			
w3af	✓			
Widow	✓	✓	✓	✓
ZAP				

As shown in Table VII, none of the other scanners managed to access the configuration panel. This is because phpBB requires a double login. Arachni, jÄk, Skipfish, w3afand ZAP all require user-supplied credentials together with parameters before running. Based on the traces they do not try these credentials on the admin login form, only the first login form. Enemy of the State, on the other hand, tries the standard username and password `scanner1`. This was enough to log in but it did not manage to log in as an admin.

Our scanner solves the double login by being consistent with the values we submit. This allows us to both authenticate as a user and then also as an admin when presented with the login prompt. After submitting the form in configuration panel with our taint tokens and later revisiting it, we detect the inter-state dependency and can fuzz the source and sink.

TABLE VII: Steps to recreate the vulnerability in phpBB. The columns contain the file name and line of code for each step.

Crawler	admin/ index.php#28	admin_ board.php#34	admin_ board.php#74	admin_ board.php#34	Exploit
Arachni					
Enemy					
jÄk					
Skipfish					
w3af					
Widow	✓	✓	✓	✓	✓
ZAP					

E. Features Attribution

In this section, we identify and attribute the key features that contributed to finding the vulnerabilities in the web applications.

In particular, we try to determine the impact of our modeling, traversing and inter-state dependency analysis techniques. Below are the definitions we use in Table VIII.

a) Modeling: Modeling is considered to contribute if a combination of HTML forms and JavaScript events were used to find the code injection.

b) Traversal: Workflow traversal contributes if the point of injection depends on a previous state. This could, for example, be a form submission, a click of a button or some other DOM interaction.

c) Inter-state dependency: A code injection is defined to need inter-state dependency analysis if the point of reflection is different from the point of injection.

Table VIII shows the 25 unique code injections from the evaluation. Of these, modeling contributed to 4, workflow traversal contributed to 9, and inter-state dependency analysis contributed to 13. In total, at least one of them was a contributor in 16 unique injections. The remaining 9 were usually simpler. Four of them were from WackoPicko where the results of injection were directly reflected. SCARF had 3 directly reflected injections and osCommerce had 2. It is clear, especially for unique vulnerabilities, that modeling, workflow traversal and inter-state dependency analysis plays an important role in detecting stored XSS vulnerabilities.

TABLE VIII: For each of the vulnerabilities we note contributing features, i.e. modeling, workflow reproduction or inter-state dependency (ISD) analysis. We also present if they were uniquely detected by Black Widow.

Id	Application	Description	Model	Workflow	ISD	Unique
1	HotCRP	User upload	✓	✓		✓
2	osCommerce	Review rating				✓
3	osCommerce	Tax class				✓
4	phpBB	Admin ranks			✓	✓
5	phpBB	Configuration			✓	✓
6	phpBB	Site name			✓	✓
7	PrestaShop	Date	✓	✓		✓
8	SCARF	Add session		✓	✓	✓
9	SCARF	Comment		✓	✓	✓
10	SCARF	Conference name				
11	SCARF	Edit paper		✓	✓	✓
12	SCARF	Edit session				
13	SCARF	Delete comment		✓	✓	✓
14	SCARF	General options				
15	SCARF	User options			✓	
16	Vanilla	Comment draft			✓	✓
17	Vanilla	Locale			✓	✓
18	Vanilla	Title banner	✓			✓
19	WackoPicko	Comment				
20	WackoPicko	Multi-step		✓	✓	✓
21	WackoPicko	Picture				
22	WackoPicko	Search				
23	WackoPicko	SQL error				
24	WordPress	Comment		✓	✓	✓
25	WordPress	Nearby event	✓	✓	✓	✓

F. Missed by Us

Out of the 25 unique injections found by all scanners, we also find all 25. There was, however, an instance where Arachni found a vulnerability by injecting a different parameter than we did. This does not constitute a unique vulnerability due to our clustering, which we explain in Section IV-D. On SCARF, input elements can be dynamically generated by adding more users. The input names will simply be `1_name`, `2_name`, etc. Arachni managed to add multiple users by randomizing email addresses. Since our crawler is focused on consistency, we do not generate valid random email addresses and could therefore not add more than one user.

The drawback, as we have discussed is that is it easier to lose the state if too much randomness is used. A possible solution to this could be to keep two sets of default values and always test both when possible. There is still the risk that using multiple users can result in mixing up the state between them. It would also introduce a performance penalty as multiple submissions for each form would be required.

The w3af scanner was able to find a reflected version of a vulnerable parameter that we considered to be stored. In this particular case on SCARF, it was possible to get a direct reflection by submitting the same *password* and *retype password* in the user settings. This is what w3af did. Our scanner injected unique values into each field, resulting in an error without reflection, however, the fields were still stored. Inter-state dependency analysis was used to detect these stored values when revisiting the user settings.

Further possible improvements include updating our method for determining safe requests and more robust function hooking. A machine learning approach, such as Mitch [6], could be used to determine if a request can be considered safe. The function hooking could be done by modifying the JavaScript engine instead of instrumenting JavaScript code.

G. Vulnerability Exploitability

For the six new vulnerabilities, we further investigate the impact and exploitability. While all of these vulnerabilities were found using an admin account in the web application, the attacker does not necessarily need to be an admin. In fact, XSS payloads executed as the admin gives a higher impact as the JavaScript runs with admin privileges. What the attacker needs to do is usually to convince the admin to click on a link or visit a malicious website, i.e. the attacker does not require any admin privileges. Although, there might be an XSS vulnerability in the code, i.e. user input being reflected, there are orthogonal mitigations such as CSRF tokens and CSP that can decrease the exploitability.

To exploit the HotCRP vulnerability the attacker would have to guess a CSRF token, which is considered difficult. Similarly, PrestaShop has a persistent secret in the URL which would have to be known by the attacker. One of the WordPress vulnerabilities was a self-XSS, meaning the admin would need to be convinced to, in this case, input our payload string, while the other one required a CSRF token. Finally, osCommerce required no CSRF tokens making it both high impact and easy to exploit.

H. Coordinated Disclosure

We have reported the vulnerabilities to the affected vendors, following the best practices of coordinated disclosure [15]. Specifically, we reported a total of six vulnerabilities to HotCRP, osCommerce, PrestaShop and WordPress.

So far our reports have resulted in HotCRP patching their vulnerability [24]. A parallel disclosure for the same vulnerability was reported to PrestaShop and is now tracked as CVE-2020-5271 [1]. Due to the difficulty of exploitation, WordPress did not consider them vulnerabilities. However, the *nearby event* vulnerability is fixed in the latest version. We have not received any confirmation from osCommerce yet.

VI. Related Work

This section discusses related work. Automatic vulnerability scanning has been a popular topic due to its complexity and practical usefulness. This paper focuses on blackbox scanning, which requires no access to the application's source code or any other input from developers. We have evaluated our approach with respect to both community-developed open-source tools [36, 34, 28] and academic blackbox scanners [30, 8]. There are also earlier works on vulnerability detection and scanning [2, 17, 35, 23, 10, 11]. While we focus on blackbox testing, there is also progress on whitebox security testing [13, 18, 22, 25, 39].

As previous evaluations [3, 9, 37, 40, 29] show, detecting stored XSS is hard. A common notion is that it is not the exact payload that is the problem for scanners but rather crawling deep enough to find the injections, as well as, model the application to find the reflections. Similar to our findings, Parvez et al. [29] note that while some scanners were able to post comments to pictures in WackoPicko, something which requires multiple actions in sequence, none of them was able to inject a payload.

We now discuss work that addresses server-side state, client-side state, and tracking data dependencies.

a) Server-side state: Enemy of the State [8] focuses on inferring the state of the server by using a heuristic method to compare how requests result in different links on pages. Black Widow instead takes the approach of analyzing the navigation methods to infer some state information. For example, if the previous edge in the navigation graph was a form submission then we would have to resubmit this form before continuing. This allows us to execute sequences of actions without fully inferring the server-side state.

One reason many of the other scanners pay little attention to server-side state is to prioritize performance from concurrent requests. Skipfish [42] is noteworthy for its high performance in terms of requests per second. One method they use to achieve this is making concurrent requests. Concurrent requests can be useful in a stateless environment since the requests will not interfere with each other. ZAP [28], w3af [34] and Arachni [36] take the same approach as Skipfish and use concurrent requests in favor of better state control. Since our traversing method relies on executing a sequence of possibly state-changing action we need to ensure that no other state-changing requests are sent concurrently. For this reason, our approach only performs actions in serial.

b) Client-side state: jÄk considers client-side events to improve exploration. The support for events is however limited, leaving out such events as form submission. While other scanners like Enemy of the State, w3af, and ZAP execute JavaScript, they do not model the events. This limits their ability to explore the client-side state. As modern applications make heavy use of JavaScript, Black Widow offers fully-fledged support of client-side events. In contrast to jÄk, Black Widow models client-side events like any other navigation method. This means that we do not have to execute the events in any particular order which allows us to chain them with other navigations such as form submissions.

c) Tracking data dependencies: Tracking payloads is an important part of detecting stored XSS vulnerabilities. Some scanners, including Arachni, use a session-based ID in each payload. Since the ID is based on the session this can lead to false positives as payloads are reused for different parameters. jÄk and Enemy of the State use unique IDs for their payload but forgets them on new pages. w3af uses unique payloads and remembers them across pages. ZAP uses a combination in which a unique ID is sent together with a generic payload but in separate requests. This works if both the ID and payload are stored on a page. In addition to using unique IDs for all our payloads, Black Widow incorporates the inter-state dependencies in the application to ensure that we can fuzz the correct input and output *across* different pages.

LigRE [10], and its successor KameleonFuzz [11] use a blackbox approach to reverse engineering the application and apply a genetic algorithm to modify the payloads. While they also use tainting inside the payloads to track them, we use plaintext tokens to avoid filters destroying the taints. While Black Widow works on live applications, KameleonFuzz requires the ability to reset the application. Unfortunately, neither LigRE nor KameleonFuzz are open-source, which has hindered us from their experimental evaluation.

VII. Conclusion

We have put a spotlight on key challenges for crawling and scanning the modern web. Based on these challenges, we have identified three core pillars for deep crawling and scanning: navigation modeling, traversing, and tracking inter-state dependencies. We have presented Black Widow, a novel approach to blackbox web application scanning that leverages these pillars by developing and combining augmented navigation graphs, workflow traversal, and inter-state data dependency analysis. To evaluate our approach, we have implemented it and tested it on 10 different web applications and against 7 other web application scanners. Our approach results in code coverage improvements ranging from 63% to 280% compared to other scanners across all tested applications. Across all tested web applications, our approach improved code coverage by between 6% and 62%, compared to the *sum* of all other scanners. When deployed to scan for cross-site scripting vulnerabilities, our approach has featured no false positives while uncovering more vulnerabilities than the other scanners, both in the reference applications, i.e. phpBB, SCARF, Vanilla and WackoPicko, and in production software, including HotCRP, osCommerce, PrestaShop and WordPress.

Acknowledgments: We would like to thank Sebastian Lekies for inspiring discussions on the challenges of web scanning. We would also like to thank Nick Nikiforakis and the reviewers for their valuable feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

[1] CVE-2020-5271. Available from MITRE, CVE-ID CVE-2020-5271., April 20 2020.

[2] 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 *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[3] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[4] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 81–90. ACM, 2009.

[5] Bugcrowd. The State of Crowdsourced Security in 2019. https://www.bugcrowd.com/, 2020.

[6] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 528–543. IEEE, 2019.

[7] Adam Doupé. Wackopicko, 2018.

[8] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security Symposium 12*, pages 523–538, 2012.

[9] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny cant pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[10] 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 *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 252–261. IEEE, 2013.

[11] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48, 2014.

[12] Facebook. A Look Back at 2019 Bug Bounty Highlights. https://www.facebook.com/notes/facebook-bug-bounty/a-look-back-at-2019-bug-bounty-highlights/3231769013503969/, 2020.

[13] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[14] Roy Fielding and Jiulian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, June 2014.

[15] Google. Project zero: Vulnerability disclosure faq, 2019.

[16] Google. Vulnerability Reward Program: 2019 Year in Review. https://security.googleblog.com/2020/01/vulnerability-reward-program-2019-year.html, 2020.

[17] William GJ Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Penetration testing with improved input vector identification. In *2009 International Conference on Software Testing Verification and Validation*, pages 346–355. IEEE, 2009.

[18] 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 *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

[19] SE Idrissi, N Berbiche, F Guerouate, and M Shibi. Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. *International Journal of Applied Engineering Research*, 12(21):11068–11076, 2017.

[20] InfoSecurity. XSS is Most Rewarding Bug Bounty as CSRF is Revived. https://www.infosecurity-magazine.com/news/xss-bug-bounty-csrf-1-1-1-1/, 2019.

[21] Security Innovation. Google Awards $1.2 Million in Bounties Just for XSS Bugs. https://blog.securityinnovation.com/google-awards-1.2-million-in-bounties-just-for-xss-bugs, 2016.

[22] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[23] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006.

[24] Eddie Kohler. Correct missing quoting reported by Benjamin Eriksson at Chalmers. https://github.com/kohler/hotcrp/commit/81b7ffee2c5bd465c82acf139cc064daacca845c, 2020.

[25] Xiaowei Li, Wei Yan, and Yuan Xue. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 25–36, 2012.

[26] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE, 2008.

[27] Hrvoje Niki. Wget - gnnu project, 2019.

[28] OWASP. Owasp zed attack proxy (zap), 2020.

[29] 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 *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 186–191. IEEE, 2015.

[30] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *International Symposium on Recent Advances in Intrusion Detection*, pages 295–316. Springer, 2015.

[31] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic

analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[32] PortSwigger. Burp Scanner - PortSwigger. https://portswigger.net/burp/documentation/scanner, 2020.

[33] Derick Rethans. Xdebug - debugger ad profiler tool for php, 2019.

[34] Andres Riancho. w3af - open source web application security scanner, 2007.

[35] T. S. Rocha and E. Souto. Etssdetector: A tool to automatically detect cross-site scripting vulnerabilities. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 306–309, Aug 2014.

[36] Sarosys LLC. Framework - arachni - web application security scanner framework, 2019.

[37] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[38] The OWASP Foundation. Owasp top 10 - 2017, 2017. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[39] Alexandre Vernotte, Frédéric Dadeau, Franck Lebeau, Bruno Legeard, Fabien Peureux, and François Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In Atul Prakash and Rudrapatna Shyamasundar, editors, *Information Systems Security*, pages 358–377, Cham, 2014. Springer International Publishing.

[40] Marco Vieira, Nuno Antunes, and Henrique Madeira. Using web security scanners to detect vulnerabilities in web services. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 566–571. IEEE, 2009.

[41] WHATWG. Html standard, 2019.

[42] Michal Zalewski. Skipfish, 2015.

VIII. Appendix

A. Scanner configuration

1) Arachni: The following command was used to run Arachni.

```
arachni [url] --check=xss* --browser-cluster-pool
    -size=1 --plugin?autologin:url=[loginUrl],
    parameters="[userField]=[username]&[
    passField]=[password]",check="[logout string
    ]}
```

2) Black Widow: The following command was used to run Black Widow.

```
python3 crawl.py [url]
```

3) Enemy of the State: First we changed the username and password in the web application to *scanner1* then we ran the following command.

```
jython crawler2.py [url]
```

4) jÄk: We updated the `example.py` file with the URL and user data.

```
url = [url]
user = User("[sessionName]", 0, url, login_data =
     {"[userField]": "[username]", "[passField]"
    : "[password]"}, session="ABC")
```

5) Skipfish: The following command was used to run Skipfish.

```
skipfish -uv -o [output]
         --auth-form [loginUrl]
         --auth-user-field [userField]
         --auth-pass-field [passField]
         --auth-user [username]
         --auth-pass [password]
         --auth-verify-url [verifyUrl]
         [url]
```

6) w3af: For w3af we used the following settings, *generic* and *xss* for the audit plugin, *web_spider* for crawl plugin and *generic* (with all credentials) for the auth plugin.

7) Wget: The following command was used to run Wget.

```
wget -rp -w 0 waitretry=0 -nd --delete-after --
    execute robots=off [url]
```

8) ZAP: For ZAP we used the *automated scan* with both *traditional spider* and *ajax spider*. In the *Scan Progress* window we deactivated everything that was not XSS. Similar to Enemy of the State, we changed the credentials in the web application to the scanner's default, i.e. *ZAP*.