YURASCANNER: Leveraging LLMs for Task-driven Web App Scanning

Aleksei Stafeev, Tim Recktenwald, Gianluca De Stefano, Soheil Khodayari, Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
{aleksei.stafeev, tim.recktenwald, gianluca.de-stefano, soheil.khodayari, pellegrino}@cispa.de

***Abstract*—Web application scanners are popular and effective black-box testing tools, automating the detection of vulnerabilities by exploring and interacting with user interfaces. Despite their effectiveness, these scanners struggle with discovering deeper states in modern web applications due to their limited understanding of workflows. This study addresses this limitation by introducing YURASCANNER, a task-driven web application scanner that leverages large-language models (LLMs) to autonomously execute tasks and workflows.**

YURASCANNER operates as a goal-based agent, suggesting actions to achieve predefined objectives by processing webpages to extract semantic information. Unlike traditional methods that rely on user-provided traces, YURASCANNER uses LLMs to bridge the semantic gap, making it web application-agnostic. Using the XSS engine of Black Widow, YURASCANNER tests discovered input points for vulnerabilities, enhancing the scanning process's comprehensiveness and accuracy.

We evaluated YURASCANNER on 20 diverse web applications, focusing on task extraction, execution accuracy, and vulnerability detection. The results demonstrate YURASCANNER's superiority in discovering new attack surfaces and deeper states, significantly improving vulnerability detection. Notably, YURASCANNER identified 12 unique zero-day XSS vulnerabilities, compared to three by Black Widow. This study highlights YURASCANNER's potential to revolutionize web application scanning with its automated, task-driven approach.

I. INTRODUCTION

Web application scanners represent the quintessence of black-box testing tools as they explore the attack surface of web applications and detect vulnerabilities in an automated manner. Starting from a seed webpage, they iteratively search for forms and URLs by interacting with the user interfaces. Then, they inject special inputs into these forms and URLs to search for vulnerabilities. While web scanners have been proven effective in the detection of web vulnerabilities, they still perform poorly or even fail in automatically discovering deeper states, leaving potential vulnerabilities undetected.

One of the main shortcomings of web scanners is their unawareness of the tasks and workflows that characterize modern web applications. These workflows require web scanners to understand the sequence of various steps and execute them in the correct order. For example, a scanner needs to add products to a shopping cart before proceeding with the payment. Unfortunately, existing web scanners (e.g., [1]–[3]) rely on basic page navigation strategies, such as breadth-first and random strategies [4], making it unlikely for a scanner to execute the correct sequence of actions. Recently, model-based methods have been proposed to guide crawlers through workflows. These models can be state machines learned during crawling (e.g., [5], [6]), user-provided traces that a crawler can replay as-is (e.g., [7]), or reinforcement learning techniques that learn general navigation patterns from user traces (e.g., [8]). Unfortunately, these approaches do not scale well because models and traces are not easily transferable across different web applications, and web applications can contain tens or hundreds of workflows, making their manual generation challenging. Recently, non-academic approaches have proposed using large language models (LLMs) to assist users during interactions with websites, such as Natbot [9]. However, we found that these approaches often fail to complete web application workflows because their actions are too fine-grained and decisions are based solely on the current page. This makes them unsuitable for autonomously exploring a web application in a task-driven manner.

In this paper, we address the challenge of creating a task-driven web application scanner capable of executing workflows in a web application-agnostic manner, and present one of the first automated tools for this purpose called YURASCANNER. Our approach models the scanner as a goal-based, rational agent [10] instead of a reactive agent (i.e., [9]), which, given an objective (the task) and a history of previous actions, suggests the next action to bring the scanner closer to its objective. Instead of building a model to select the next action, we use LLMs based on the intuition that LLMs are trained on publicly available documents describing workflows and functionalities of web applications. We present a method to represent actions, states, and objectives at a semantic level aligned with these documents and query an LLM to guide exploration towards the objectives. To bridge the semantic gap between the real web application and the LLM, we employ sensors and actuators to process webpages, extract semantic information, and execute the proposed actions in a real browser. To demonstrate the advantages of a task-driven web application scanner in a security context, we integrated the vulnerability detection engine of Black Widow [3] into YURASCANNER to detect cross-site scripting (XSS) vulnerabilities using the forms discovered through our task-driven exploration.

We comprehensively evaluated YURASCANNER against 20 real and diverse web applications, focusing on task extraction and execution accuracy, a comprehensive measure-

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240388
www.ndss-symposium.org

ment of the attack surface discovered and its characterization, and an assessment of the vulnerability detection benefits of our task-driven crawling approach. We compared our results against state-of-the-art baseline techniques, specifically the Black Widow scanner, and BFS and Random BFS navigation strategies, yielding four main takeaways.

First, the task generation process produced 77% valid tasks (1,818 out of 2,361 tasks). Of these 1,818 tasks, YURASCANNER successfully executed 1,115 (61.3%), completing 667 tasks to the last step and 448 tasks to the last but one step. The remaining 703 task executions failed primarily due to misalignments between expected and existing states (75%) and incorrect actions (25%).

Second, YURASCANNER excels in identifying significant new attack surfaces despite generally discovering fewer unique aspects compared to other methods, which have a broader, less focused approach. Specifically, these new discoveries constitute a substantial portion: 35.4% (Forms) and 25.7% (URLs) of the total attack surface discovered jointly by both YURASCANNER and baseline tools, reflecting an average increase of 55.19% (Forms) and 35.16% (URLs).

Third, the newly-discovered attack surfaces by YURASCANNER are deeper than those discovered by other tools, ranging from 41.5% at maximum depths of three steps to 14.3% at depths up to 15 steps. These forms were not discovered by other tools.

Lastly, our findings demonstrate YURASCANNER's superior performance in vulnerability detection over Black Widow. Notably, YURASCANNER and Black Widow found 13 unique, zero-day XSS vulnerabilities across three applications: *Redacted*, Moodle, and Leantime, with YURASCANNER discovering 12 new vulnerabilities compared to the three vulnerabilities found by Black Widow.

In summary, this paper makes the following contributions:

- We propose one of the first fully automated task-driven web application scanning techniques, modeled as a goal-driven agent that can complete multi-step tasks and follow complex workflows autonomously, leveraging a large-language model.
- We implemented our technique in a tool called YURASCANNER, which we evaluated against 20 real and diverse web applications.
- We comprehensively assessed the task generation and execution with an extensive manual review of all 2,361 tasks generated on ten randomly selected web applications.
- We comprehensively measured and characterized the new attack surface discovered by YURASCANNER and compared it to three baseline approaches (Black Widow, BFS, and Random BFS) across ten applications.
- We demonstrated how the new attack surface benefits vulnerability detection by comparing YURASCANNER with Black Widow, identifying 13 new zero-day XSS vulnerabilities, with YURASCANNER uncovering 12 vulnerabilities compared to the three found by Black Widow.

II. BACKGROUND

In this section, we first introduce web application scanners and how they operate in Section II-A, and then we describe the challenges they encounter in Section II-B.

A. Web Scanners Architecture

Web scanners [4], [11] are dynamic security testing tools that identify vulnerabilities within web applications by examining them during runtime. These tools typically have two components: (i) a web crawler that navigates through webpages, interacting with different elements such as links, forms, and buttons, with the overall goal of identifying all webpages and their input points, such as forms, request endpoints and their parameters; and (ii) an attack module that tests the discovered endpoints and forms using a dictionary of attack inputs, monitoring the application's responses to identify program executions that trigger vulnerabilities. Contrary to static analysis methods, which detect vulnerabilities by analyzing the source code without running it, security scanners identify vulnerabilities that only manifest during program execution, resulting in little-to-no false positives and a PoC exploitation input.

B. Crawling is a Key Challenge

Existing web scanners employ workflow-agnostic crawling algorithms [4], where the next action is not selected based on the application logic of the visited pages. For example, state-of-the-art security scanners like Black Widow [3] use a randomized BFS approach, which selects the next action randomly from the available options. Another popular example includes crawlers, such as Arachni [12], ZAP [13], and w3af [14], that use a BFS approach by storing new URLs in a queue with a first-in-first-out (FIFO) policy. These strategies seldom result in the appropriate sequence of steps needed for the crawler to reach deeper application states.

Consider, for example, the web UI of *Redacted* in Figure 1 and the steps to create a coupon that YURASCANNER discovered to suffer from an XSS vulnerability. To identify the vulnerability, a crawler needs to execute six steps: clicking through the `Tools` link (step 1), the `Entities` link (step 2), the `Entity Management` link (step 3), the `Create` button (step 4), filling out the form (step 5), and finally clicking the `Save` button (step 6). The input field `Parameter B` (step 5) is neither server-side validated nor sanitized, resulting in an XSS vulnerability. Traditional security scanners cannot detect this vulnerability because their crawlers cannot perform these six steps in the correct order, thus failing to reach and test this field.

III. PROBLEM STATEMENT

In this paper, we tackle the challenge of handling multi-step tasks and complex workflows in web applications, with the overarching goal of studying if such task-driven crawling can identify unknown security vulnerabilities. Specifically, we answer the following research questions:

RQ1: Automated Extraction of Tasks. How can we identify a comprehensive list of tasks and workflows present in a given web application? The first part of our study aims to answer

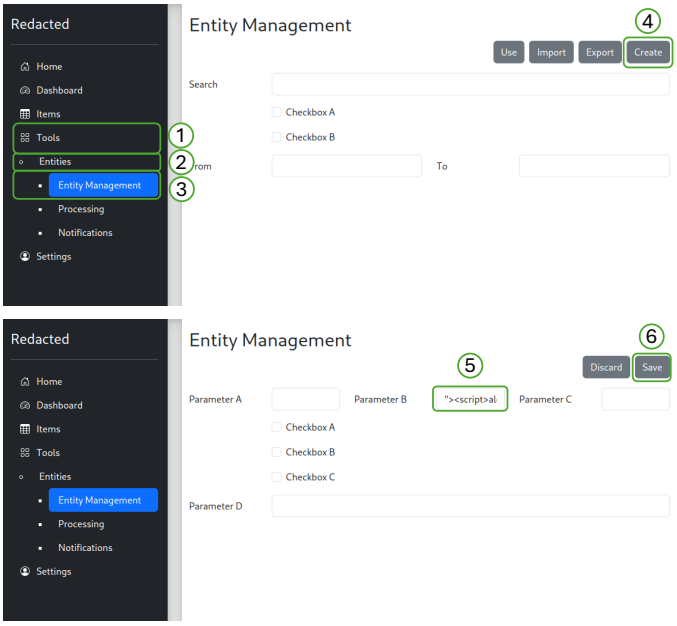

Fig. 1: Overview of a real XSS vulnerability we discovered in *Redacted*.

this RQ. We develop a technique to extract relevant workflows starting from a single seed URL and performing a shallow crawl to gather primary webpages of the application, extracting text content including menus, links, and buttons.

We then process this content using LLMs to understand the semantic functionalities offered by the application. Finally, we organize the extracted functionalities into a catalog of tasks that a web agent can perform in subsequent stages.

RQ2: Automated Execution of Tasks. Given a task, how can we visit a web application and perform the task automatically? Current crawling techniques often struggle to reach deep application states that require complex interactions, such as multi-step forms and nested functionalities like checkout processes in shopping cart applications.

We address this problem by treating the crawler as a goal-driven agent that performs actions based on the current state and past states. Specifically, we leverage the advanced semantic understanding capabilities of LLMs, using one-shot prompting with history, to identify which action to perform next in a given workflow and webpage, ultimately interacting with the application in a more human-like manner. The key insight of our approach is that LLMs like GPT-4 are trained on documentation files of a plethora of web applications, thereby they likely possess a foundational knowledge of common application functionalities and how they are structured and operate, which enhances their ability to accurately interpret and navigate complex workflows.

RQ3: Attack Surface Coverage. What are the impacts of task-driven crawling on attack surface coverage? In this RQ, we aim to quantify improvements or implications in attack surface coverage offered by a task-driven scanner by comparing it to state-of-the-art scanners like Black Widow [3]. Similarly to prior work [2]–[4], we use common coverage metrics, such as URLs and forms, analyzing the unique and shared attack surface coverage offered by each approach. Overall, we show that our approach increases the coverage of web scanners in areas that are hard to reach, including deep and complex application states, leading to more comprehensive testing.

RQ4: Characterization of the Attack Surface. How does the discovered attack surface correlate with the depth and complexity of application states during task execution? While RQ3 measures the size of the attack surface, this RQ investigates how the discovered attack surface correlates with the ability to explore deeper web application states. Specifically, we aim to understand how much of the attack surface is uncovered as the steps of a task are executed, where state-sensitive operations are located, and how accessible they are to security tests. By analyzing the complexity of the tasks our scanner completes and considering the number of intermediate steps taken, we can analyze the effort required to uncover vulnerabilities located in the hard-to-reach attack surface, which often requires specific conditions or user interactions to be exposed.

RQ5: Vulnerability Detection. Is our task-driven exploration approach helpful for vulnerability detection? In this RQ, we want to determine whether our approach can uncover new vulnerabilities that traditional security scanners might miss. By enabling the web agent to navigate deep application states, we hypothesize that it will identify security flaws in areas typically overlooked by conventional scanners, such as insecure payment handling in e-commerce checkouts or insufficient input validation in multi-step forms. In addition, understanding the context in which web elements operate is crucial for accurate vulnerability detection. For instance, identifying whether a text input field is part of a form or a search box can significantly influence the type of security tests performed. Current tools often lack the semantic understanding needed to accurately interpret and interact with web elements. By comparing the results of our task-driven scanning with traditional methods, we aim to demonstrate its efficacy in providing more comprehensive security assessments and uncovering previously unknown vulnerabilities.

IV. YURASCANNER

This section presents YURASCANNER, and the three main components, i.e., the way YURASCANNER executes a given task (Section IV-A), the way YURASCANNER gathers potential tasks of a given application (Section IV-B), and the way security testing can be interwoven with YURASCANNER's navigation strategy (Section IV-C). Figure 2 shows the overview of our approach.

A. Task Execution

This section presents our approach when executing a given task t. We start presenting the main design decisions and the overview of our approach (Section IV-A1). Then, we review each component of our architecture (Sections IV-A2 to IV-A4).

1) Design Decisions and Approach Overview: Executing a task on a web application means correctly selecting the sequence of UI actions, e.g., mouse clicks or form submissions. Each action will bring the scanner closer to completing the task t. This problem can be conveniently modeled as a *rational,*

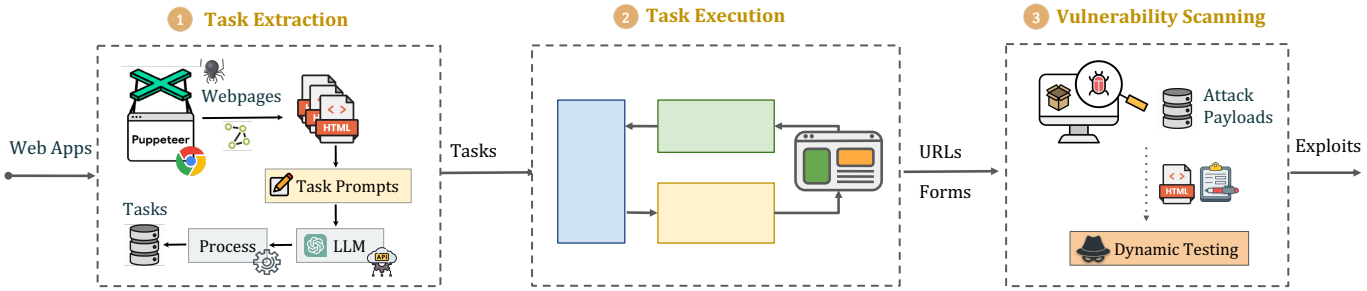

Fig. 2: Overview of YURASCANNER.

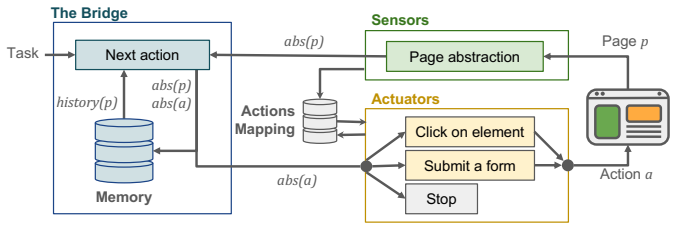

Fig. 3: Design of the task-driven crawler component of YURASCANNER modeled as a rational, goal-based agent.

goal-based agent [10]. However, implementing a rational goal-based agent requires addressing two key challenges: (1) *How are decisions taken?* and (2) *What is the right abstraction level for an agent?*.

Design Decisions — Prior work in web exploration has proposed reinforcement learning (RL) approaches to create such agents where they can determine the next action based on the current and past states. One line of work proposed deriving a state-action mapping through a trial-and-error learning strategy and an appropriate reward function, e.g., [6]. Unfortunately, these approaches result in application-specific or workflow-specific navigation strategies that do not transfer. Another line of work proposed using user-provided examples (state-action mappings extracted from user-generated click streams) to indicate possible actions to an agent, e.g., [8]. Unfortunately, these examples are hardly available in practice, rendering these approaches inadequate for our purposes. As an alternative, examples may be extracted from publicly available documents, such as user manuals and developer documentation, describing the expected sequence of actions. Unfortunately, these documents are written in natural language and the extraction of actions may be impractical.

Our approach is based on the intuition that if we could represent actions, states, and objectives at a semantic level closer to the descriptions of these documents, we could use a language model to predict the next action as the next most probable sequence of tokens. In addition, as documentation about web applications and their workflows are publicly available, current large-language models may have been already trained on them, sparing us from running expensive operations such as model training or fine-tuning. To further support our intuition, a recent GitHub project showed the viability of controlling a browser through an LLM, i.e., Natbot [9], to assist users in navigating webpages. However, we verified that Natbot hardly completes web application workflows because (i) the actions are too tailored for browser interaction (e.g., scroll up and down) and (ii) the model suffers from the endless loop crawling problem, making this approach unfit to autonomously explore in a task-driven manner a web application. Accordingly, we revisited the template of Natbot and performed additional experiments using OpenAI GPT-4, considering different abstraction levels when integrating pages and actions in the LLM prompt, e.g., removing unnecessary tags, keeping textual descriptions of actions (e.g., anchors and forms), introducing the concept of memory, and engineering the prompt to keep the model focussed throughout the execution. Our results showed the viability of this approach. We discuss the technical details of our choices in the rest of this section.

Overview — Figure 3 shows the high-level architecture of the crawler component of YURASCANNER. We model it as an agent with three main modules. The *Bridge* is the module that, given an appropriate description of the task, the current state (an abstract page), and, if any, previously visited pages, proposes the next action among the ones available on the current abstract page. The *Sensors* module is responsible for analyzing the current page through a browser instance (e.g., Puppeteer) and creating an abstract page containing the available actions. The *Actuators* module is responsible for executing the action selected by the Bridge. Actuators are connected to the same browser instance as the sensors and can perform actions such as clicking on an HTML element or filling out and submitting a form.

Given a task t, the three modules sensors, the bridge, and actuators, are iteratively executed whenever the agent visits one page and until the bridge issues a stop action. Below, we present each module in that order.

2) Sensors: As soon as the browser loads a new page, whether it is the initial page or a new page after the actuators perform an action, the sensors take a snapshot of the rendered DOM page p and generate a simplified page $abs(p)$ retaining the semantics of the available actions in p.

Actions — The module identifies all HTML elements a user interacts with when navigating a page. First, it gathers all anchor tags, buttons, inputs of type “button” or “submit,” and elements containing the `onclick` attribute using CSS query selectors. However, modern web applications increasingly rely on event handlers attached to HTML elements, which would be missed by only considering a limited set of static HTML

clickables. Thus, we opted to reuse the event handler collection scripts by jÄk [2] and Black Widow [3], which rely on JavaScript function hooking of the event registration APIs of the `HTMLElement` object. We modified the function hooking routine so that we collect the HTML element where the handler is registered. Finally, we identify all HTML forms via CSS selectors.

The collected HTML elements may contain items that are not displayed, such as those with visibility set to false, or elements not visible due to being occluded by other HTML elements, e.g., those with a low z-index value. These elements are not relevant to the logic of the workflow and may introduce noise when processed by the bridge. Accordingly, we remove them from the list. We also remove elements that either have an empty or overly large textual representation. An empty label does not allow the bridge to infer any information about an element's functionality, therefore, the corresponding element cannot be used meaningfully. On the other hand, overly long strings have to be omitted in order to stay within the token limit of the bridge.

Action Mapping — The output of the previous step is a list of JavaScript `HTMLElement` instances, one for each HTML tag representing an action. We associate each object with a unique per-page identifier, which is an incremental integer. The mapping object-ID is stored in the Actions Mapping. This mapping is later used by the actuators to re-identify the HTML element starting from its ID.

Semantics — Next, we identify text strings that can capture the semantics of the actions implemented via these HTML elements. We do this by searching for the first non-empty strings in tag attributes and properties in the following order: accessibility attributes (e.g., the `aria-label` attribute), attributes with advisory information about an element (e.g., the `title` attribute), the text contained within an HTML element (e.g., the `innerText` property), and the displayed default attribute of tags (e.g., the `value` attribute).

We handle form elements similarly by querying a subset of attributes and inserting them into a generic template. First, we include the `name` attribute, which might convey the purpose of the form. If the `name` is undefined, we instead use the `id` value as a fallback. We also keep the original `action` and `method` attributes intact. The naming scheme of the endpoint may provide additional context on the form's intended use (e.g., `action="/advanced_search.php"`), whereas the HTTP method indicates the likeliness of the request being state-changing.

The process of eliminating any other attributes of an element or form keeps the textual representation concise and reduces the amount of information that has to be processed by the bridge.

Abstract Action and Page — For each action a, we create an abstract action $abs(a)$, which is an HTML-like representation of the original tag. This abstract action contains the same HTML tag name, the unique identifier set as the `id` attribute, and the semantic string. The abstract page is the URL of the page, the title of the HTML page, and the list of the abstract actions.

3) The Bridge: The bridge is responsible for the decision-making at each step of a task execution. The bridge uses an LLM to predict the next action. This prediction is made using the prompt shown in Figure 4, which we engineered using a one-shot [15] prompting technique and persona assignment [16] to increase focus and task compliance.

We divide the prompt into three sections. In the first section, we assign the model a persona, i.e., "You are Yura, an agent controlling a web browser," to improve instruction compliance. This section includes the task instructions the model is expected to execute, the content of the abstract page (i.e., URL, title, and HTML-like actions), and the types of commands the model can issue, such as "click" or "fill and submit a form", along with the ID of the action. If the model completes the task or is unable to continue, it can issue a special command called "stop", which will halt the task execution.

The second section of the prompt contains one example of an interaction, including an example of an abstract page, previous actions, and an appropriate next action. To minimize syntactically incorrect commands, we expanded this section with a reminder of the possible command types, i.e., click, fill and submit, or stop. The third part of the prompt is the actual query for the LLM, which includes the current abstract page and the previous actions (retrieved from the local memory). We observed that the model tends to lose focus after issuing a few consecutive commands. Accordingly, we start the third section with a reminder about the model's persona and its role, i.e., "Remember, you are Yura, an agent controlling a web browser."

After the model selects a command and the ID, the bridge stores the selected action and page in the local memory. We empirically tested on a few examples that a history of six previous actions provides an adequate balance between performance and accuracy.

4) Actuators: The actuators are responsible for executing the abstract action selected by the bridge on the browser instance. Below, we present how the actuators operate.

STOP — If the command is `STOP`, the actuator halts the execution of the current task. As a result, a new task is selected and a new task execution loop starts.

CLICK x — If the next action is `CLICK` x, the actuator checks if the identifier x is associated with a concrete clickable in the action mapping and retrieves the `HTMLElement` JavaScript object associated with that action. Then, it fires a click event on that HTML element by invoking the `click()` function of the Puppeteer API. Anchors may have set the `target` attribute, which instructs the browser where to display the linked page, e.g., in a new tab (`_blank`) or on the current page (`_top`). Opening new tabs complicates the management of headless browsers as each tab needs to be tracked and managed separately (i.e., lifecycle management and error handling), therefore we override the `target` attribute and replace it with `_self`, which instructs the browser to reuse the current context.

After firing a click event, YURASCANNER waits for five seconds, allowing sufficient time for the browser to execute the JavaScript handler and, in the case of a top-level navigation

```
You are Yura, an agent controlling a web browser. For each step, you are given the following information:
 1. a simplified description of what's visible in the browser
 2. the URL of the current page
 3. the title of the current page
 4. the last steps that you completed
 5. the current task to achieve

You can issue these commands:
  CLICK X - click on an element with id X.
  FILL & SUBMIT FORM X - fill and submit the form with id X.
  STOP - stop when you think you cannot proceed further, e.g., a functionality is missing

The current browser content is provided in a simplified HTML-like syntax. Clickable elements such as links and buttons are represented like this:
  <button id=0>Cancel</button>
  <a id=1>Sign in</a>

Forms are provided in a similar manner:
  <form id=2 name="search" action="search/" [...]>

The last line of every prompt is as follows:
  YOUR COMMAND: (CLICK {id} / FILL & SUBMIT FORM {id} / STOP)

Whenever you see this string, issue the command that will get you the closest to achieving the current task based on the given browser information. Only give one single command per prompt. For example, you might issue the command "CLICK 0" to click on a shopping cart button with id=0 or 'FILL & SUBMIT FORM 3' to let another module fill and submit the form element with id=3. If you think the current task is finished or the task is impossible to achieve, you are expected to issue the STOP command.
```

```
THIS IS AN EXAMPLE

Current browser content:
{example_current_state}
Task: {example_task_descr}
Last completed steps: {example_last_steps}
Your command: {example_next_command}
```

```
Remember, you are Yura, an agent controlling a web browser.

Current browser content:
{current_state}
Task: {task_descr}
Last completed steps: {last_steps}
Your command:
```

Fig. 4: A simplified template showing the three main parts of the LLM prompt: the initial part is the preamble with the instructions, an example of input and expected out, and the current input.

event, to load the necessary resources to render the page when the web application under test runs on the same machine as YURASCANNER. However, the parameter is configurable and users can adjust it to account for network delays. After that, the sensors continue the execution of the task by creating an abstract page of the new page.

FILL & SUBMIT FORM x — A form submission encompasses different actions, i.e., filling each input field with a correct value and then submitting the form. We envisioned two possible strategies to handle forms. The first one requires the bridge to issue fine-grained commands such as `FILL FIELD` x and `SUBMIT FORM`. While these form-specific commands work well, they overload the LLM prompt with additional task instructions, decreasing LLM compliance with our workflow instructions and reducing the task completion rate. The second strategy splits the instructions and responsibilities between the bridge and the actuators. The bridge is responsible for making a high-level decision about the next action, i.e., `FILL & SUBMIT FORM` x. The actuator is then responsible for concretizing the action, i.e., filling the input fields and submitting the form. Our preliminary evaluation showed that the second strategy keeps the bridge more focused, increasing the number of completed tasks.

More specifically, whenever the bridge issues a fill and submit command, the actuator submits a prompt to the LLM. We design the prompt as a one-shot prompt (providing an example of a form with the desired output). A simplified version of our prompt is shown in Figure 5. Forms can also include fields other than text input boxes such as drop-down selections (e.g., country) and checkboxes (e.g., accept terms and conditions), which are clickables. When expanding the action space to fill forms with click actions, we observed that the LLM did not always confidently handle them correctly. Accordingly, we do not provide these fields to the LLM but handle them algorithmically. For checkboxes and radio elements, we click on all unchecked entries. For drop-down selections, we choose the second entry in case there is more than one possible option to avoid invalid default selections such as "Please select a country", which is commonly found as the first element of drop-down menus.

B. Tasks Catalog Generation

YURASCANNER takes as input a catalog of tasks and executes each task t as described in Section IV-A. When the bridge issues a STOP command, YURASCANNER fetches the next task and executes it. YURASCANNER stops when there are no more tasks in the list.

In general, a user can manually provide a list of tasks and feed it to YURASCANNER. However, in this paper, we intend to explore the possibility of generating such a list automatically. An ideal list of tasks should cover all workflows of a web application. Unfortunately, such a complete list does not exist in practice. One approach is to manually enumerate the workflows by identifying the entities handled by the web application, e.g., products, customers, and wiki pages, and the possible operations on them, e.g., create, delete, and modify. This approach may be impractical and may not scale to other web applications.

```
You are FormGPT, which is an agent for automatically filling out forms on a web page.
As an input, you are given a form in HTML representation. Your task is to fill out the form with fitting values.

You can issue this command multiple times (each command in a new line):
TYPE X "text" - type the provided text into the input with id X.
```

```
Here is an example:
FORM TO FILL OUT:
{example_form}
YOUR LIST OF COMMANDS: {example_commands}
```

```
The actual form follows now. Please provide your list of commands as an answer.
FORM TO FILL OUT:
{current_form}
YOUR LIST OF COMMANDS:
```

Fig. 5: A template showing the form module LLM prompt. The overall structure is similar to the bridge LLM prompt.

```
Use the following list of button labels on the website to generate a list of tasks to complete on the website. Adding items, Editing items and, finally, Deleting items. Keep tasks simple and straight to point. Like, "Add a product", "Send E-mail", "Delete a comment".
```

```
{page}
The list of tasks:
```

Fig. 6: A template showing the task generation approach.

In this paper, we propose an automated approach. We start from the seed URL of the web application under test and perform a shallow crawl of depth one to gather the primary webpages of the application, extracting text content including menus, links, and buttons. We use the sensors module to locate interactable HTML elements and keep the elements that have not appeared on previous pages to minimize the chance of generating duplicate tasks.

We then process this content using an LLM to understand the semantic functionalities offered by the application and suggest a list of tasks. We model a task as an imperative English sentence composed of a verb, e.g., "Create," and the verb's direct object, e.g., "a new user account." Recognizing that certain tasks may have dependencies (e.g., a delete or edit operation can only be executed after creating an object), we ask the LLM to sort these tasks based on basic CRUD (Create, Read, Update, Delete) operations. This structured approach ensures that YURASCANNER can navigate and interact with the application logically and efficiently, adhering to the inherent dependencies between tasks. Figure 6 shows the prompt we used for the generation.

C. Vulnerability Testing

To showcase the benefits of a task-driven web application scanner, we integrated an XSS engine with YURASCANNER. Instead of creating a new one, we selected an engine from recent works, i.e., the XSS engine of Black Widow [3]. The XSS engine uses a database of XSS payloads that call the oracle function `xss(id)` where the ID is unique for each injection.

The engine has two operational modes, i.e., safe and aggressive, and they are used as follows. For each form and for each payload, the XSS engine embeds a script that defines the oracle JavaScript function `xss(id)`. Then, it generates a unique ID, and injects the payload with the ID in all "safe" fields, i.e., text, text area, password, and email fields, then submits the form. If the injections are unsuccessful (the oracle function has not been executed), it switches to the aggressive mode where it injects the payload in other input fields, i.e., hidden fields, radio buttons, checkboxes, select fields, and upload file fields.

We combine this XSS engine with YURASCANNER as follows. First, at each iteration of the task execution (see Section IV-A), we log the URLs of web pages containing forms, the position of the forms in the pages, and the sequence of concrete actions the actuators executed to reach that URL. After the task execution, we loop over all forms. For each of them, we start from the seed URL and repeat all the concrete actions until we reach the form. Then, we run the XSS engine of Black Widow.

V. EVALUATION

This section presents the evaluation of our approach, addressing research questions RQ2-5. First, we address RQ2, where we verify the accuracy of our approach to correctly generate and execute tasks (Section V-C). Then, we address RQ3, looking at the attack surface increase when using our task-driven crawling approach, using different metrics to measure the newly discovered attack surface (Section V-D). Then, we address RQ4, looking at the depth of this attack surface, providing a characterization of the surface that YURASCANNER can now discover (Section V-E). Finally, we address RQ5, evaluating the impact of our task-driven exploration strategy in detecting vulnerabilities in these in-depth states (Section V-F).

We answer these four questions by using a testbed of real-size, modern web applications and compare our results against the results of state-of-the-art tools and techniques.

A. Implementation of YURASCANNER

We implemented YURASCANNER in JavaScript (Node.js). YURASCANNER takes as input the URL of the web application under test. Optionally, it can also take a single task or a list of tasks as input. If no task is provided, YURASCANNER generates a catalog of tasks as presented in Section IV-B.

LLM Interface — YURASCANNER interacts with an LLM through the OpenAI SDK. By changing the `base_url` configuration parameter of the SDK, YURASCANNER can also use other open-source models. LLM APIs compatible with the

OpenAI API include the Fastchat API [17][1], which enabled us to support open-source models like the Llama models using the VLLM[2] backend. When selecting the model, we considered GPT-4, a closed-source model, and three state-of-the-art open-source models, i.e., Llama3-8B [18], Gemma2-9B, and Gemma2-27B [19].

We evaluated YURASCANNER with these four models against *Redacted* and compared the success rate of the execution of 20 randomly selected tasks. GPT-4 had the highest success rate making it the best choice.

Debug Interface — Debugging a task-driven scanner is not trivial due to the stochastic behavior introduced by a model. YURASCANNER aids users with two main debug features. First, YURASCANNER can save screenshots of the rendered web pages on disk during the task execution phase. These screenshots are augmented with color-coded bounding boxes to show the actions identified by the sensors and the next action that the bridge intended to execute. Additionally, we superimpose the name of the task on the screenshot to provide full awareness to the user about the objective YURASCANNER had when reviewing them. This debug interface is fundamental in helping the user to evaluate the task execution capabilities. Furthermore, YURASCANNER supports a debug mode in the LLM interface, in which the model is replaced by the user through an interactive console. This interface helps evaluate the behavior of the sensors and actuators.

Security Tests — We integrated the XSS detection engine of Black Widow with the task-driven strategy of YURASCANNER as presented in Section IV-C.

B. Experiment Setup

We now review the configuration of our testbed, tools, and experiments.

1) Web Applications: We conducted the evaluation of this paper against real-size, modern web applications. We started with the applications selected in Black Widow [3] and removed those that are no longer updated (e.g., WackoPicko and SCARF). Then, we selected other applications from online catalogs of containerized applications. We used the Bitnami Application Catalog [20], Elestio Application Catalog [21], and DockerHub's Popular CMS Containers [22]. We selected web applications from a diverse set of categories. In total, we collected the 21 web applications shown in Table I. However, during the execution of the experiments, YURASCANNER failed when executing tasks on WordPress. The inspection of the logs revealed that WordPress included the entire documentation of the used theme in an input field. That caused the prompt to reach the maximum size allowed by the LLM, thus receiving 400 error codes from the LLM API. Accordingly, we decided to discard WordPress from our experiments.

We divided our testbed into two distinct sets. The first set contains ten randomly selected web applications that we will use to validate manually the success rates and errors in the task execution phase of our tool (i.e., TE dataset). The second set contains all web applications (i.e., the TE+VD dataset) that we will use in the vulnerability detection phase.

[1]https://github.com/lm-sys/FastChat

[2]https://github.com/vllm-project/vllm

Dataset	Name	Version	Category
TE+VD	*Redacted*	*Redacted*	*Redacted*
TE+VD	GitLab	16.11.2-ce.0	VCS
TE+VD	OpenCart	4.0.2-3	eCommerce
TE+VD	Redmine	5.1.2	Bug tracker
TE+VD	Dolibarr	19.0.2	CRM
TE+VD	Moodle	4.4.0	LMS
TE+VD	MediaWiki	1.41.1	Wiki
TE+VD	OwnCloud	10.14.0	Media sharing
TE+VD	LimeSurvey	6.5.3	Polls
TE+VD	phpBB	3.3.11	Forum
VD	NextCloud	29.0.1	Media sharing
VD	Joomla	5.1.1	CMS
VD	Leantime	3.1.4	Projects
VD	EspoCRM	8.2.5	CRM
VD	iTop	3.1.1	IT
VD	MintHCM	4.0.4	HCM
VD	Mautic	5.0.4	Marketing
VD	GLPI	10.0.15	Assets
VD	Silverpeas	6.3.5	Web Portal
VD	Monica	4.1.2	CMS
	WordPress	6.5.3	CMS

TABLE I: Web Applications.

2) Tools: We also conduct a comparative evaluation of YURASCANNER against state-of-the-art tools. Prior work [3] showed that Black Widow performed better than other scanners (academic, open source, and industrial ones) such as Arachni [12], EotS [5], jÄk [2], Skipfish [23], w3af [14], and ZAP [13]. An explanation of these performances has been suggested by recent work, showing that one of the best-performing strategies is randomized breadth-first search, as implemented by Black Widow, followed by vanilla breadth-first search, as implemented by many scanners. Based on these results, we decided to compare our tools against Black Widow as implemented in [3], for both coverage and detection, as well as against the BSF and Rand. BFS implementations in [4], for the coverage only.

3) Experiments Environment: We bootstrapped the web applications listed in Table I as Docker [24] containers in our server. We also deployed the tools in Section V-B2 and YURASCANNER as Docker containers. We configured the web application and tool containers to be on the same virtual network. At the end of each tool run, we reset the state of the web application to avoid interference between runs. We run each scanner with a maximum time limit of four hours.

To ensure fairness against the evaluated tools, we handled login and registration separately in our experiments. We supplied all tools, including YURASCANNER, with either credentials or valid session cookies and manually scripted the sequence of actions to authenticate in each app.

C. RQ2: Execution of Tasks

Methodology — The first experiment we conducted aimed to precisely measure the accuracy of generating and executing tasks against the entire VE dataset (ten web applications). We assessed accuracy by manually reviewing all task executions by reviewing the augmented screenshots created by YURASCANNER when running in debug mode. Two co-authors reviewed the screenshots, each covering the tasks of five web applications. They then cross-validated each other's work on a random sample of tasks. When labeling unclear cases, the

App	Success	Partial	Failed			Total
			Missing state	*Deviated*	*No action*	
Redacted	76	51	8	27	58	220
OpenCart	50	45	15	4	2	116
GitLab	73	31	77	0	2	183
Moodle	54	42	38	6	3	143
MediaWiki	63	26	18	30	59	196
OwnCloud	30	18	31	25	22	126
phpBB	99	79	20	28	20	246
LimeSurvey	30	38	4	9	35	116
Dolibarr	75	78	24	21	11	209
RedMine	117	40	52	22	32	263
Total	667	448	287	172	244	1,818

TABLE II: Task execution classification.

reviewers reached a consensus through discussion; the final random sampling cross-validation showed no disagreements in labeling decisions.

Reviewing a task execution required evaluating between one to fifteen screenshots (the maximum length of a task execution) per task. For each task, the reviewer first determined if the task was a valid task for the web application under test. Then, for all valid tasks, they verified whether the task was completed successfully (all steps were correctly executed), partially completed (all steps except the last one were executed correctly), or failed. When the task execution was marked as failed, the reviewer determined the reason for the error. We distinguished three types of errors:

- *Missing state* — The current state of the web application makes it impossible to execute the task unless some additional action is executed first. For example, the task *Edit a product* cannot be executed with an empty product list; the task *Add a product* should be executed successfully first.
- *Deviated* — The steps executed by the tool did not reach the target functionality of the website. For example, given a task *Add a new user to user roles*, the scanner tried to create a new role instead of adding a user to an existing one.
- *No action* — The only action issued by the tool was the `STOP` command.

Results — In total, the task generation produced 2,361 tasks. The manual review of these tasks found that 543 were invalid, meaning the functionality does not exist in the web application under test. For example, *Get detailed information about membership or subscription plans* is an invalid task for MediaWiki. Invalid task generation mainly occurred on the pages with ambiguous or insufficient context, e.g., from the login page, which only had the Login button. The MediaWiki task was generated from the page with a link to Privacy Policy page.

The review of the remaining 1,818 tasks shows that 1,115 tasks (about 61.3% of the valid tasks) were completely (667) or partially (448) executed (YURASCANNER reached the target functionality). The remaining 703 task executions failed. When examining the reasons for failure, the two most dominant errors (covering 75% of the total errors) were missing state (i.e., the LLM failed to either ensure the correct order of the tasks or establish a state with the other task) and no action (e.g., YURASCANNER issued a STOP action instead of a valid click or form command). Deviations from the expected sequence of steps accounted for about 25% of the errors. The results of the review per application are shown in Table II.

D. RQ3: Attack Surface Coverage

Methodology — We now look at the benefits of task-driven crawling in terms of attack surface coverage using the VD dataset. A common strategy to measure coverage in a black-box manner is by counting the number of unique resources collected by a crawler. Determining similarity between resources is not trivial, and prior work has shown that URL-based comparisons are a sufficiently strong signal for uniqueness. Accordingly, we selected URL-based comparisons to measure coverage. Since URLs can contain pseudo-random parameters (e.g., session identifiers or security tokens), we used a variant of the URL comparison, where we manually identified these parameter names and ignored them during the comparison. Additionally, as we are evaluating techniques to test both URLs and forms, we also include unique forms as an additional metric for coverage.

Finally, for each metric, we perform pairwise comparisons of the coverage of YURASCANNER (A) and the other tools (B). We calculate the unique attack surface discovered by one of the two tools ($A \setminus B$ and $B \setminus A$) and the attack surface discovered by both ($A \cap B$).

Results — The aggregated results across all web applications are presented in Table III. The per-application results are in Table IV.

When looking at the absolute coverage, YURASCANNER generally discovers a smaller unique attack surface than the other tools. This is likely due to the unfocused strategy followed by the other tools, which increases the chances of finding new forms and URLs. However, as we will see in Section V-E, these forms and URLs are likely located in shallow areas of web applications.

The most striking result is that the newly discovered attack surface is very significant, accounting for 35.46% (Forms) and 25.7% (URLs) of the combined attack surface discovered by both tools ($A \cup B$). This fraction amounts to an average increase of +55.19% (Forms) and +35.16% (URLs). Additionally, the interface discovered by both tools ($A \cap B$) is relatively small when compared to either of the unique surfaces discovered by each tool. On average, this shared surface is 18.3% (Forms) and 16.9% (URLs), less than half of $A \setminus B$ and $B \setminus A$. Both results indicate orthogonal capabilities, potentially covering areas of the attack surface with different inherent characteristics.

Finally, when comparing the two metrics, YURASCANNER has a slightly higher Forms-over-URLs discovery ratio, suggesting that the type of attack surface discovered by YURASCANNER has a higher density of forms.

Metric	Black Widow						BFS						Rand. BFS					
	$\setminus B$		$\cap$		$B \setminus$		$\setminus B$		$\cap$		$B \setminus$		$\setminus B$		$\cap$		$B \setminus$	
	Tot.	%	Tot.	%	Tot.	%	Tot.	%	Tot.	%	Tot.	%	Tot.	%	Tot.	%	Tot.	%
Forms	632	35.8	339	19.2	791	44.8	635	37.1	336	19.6	739	43.2	631	33.5	340	18.0	908	48.3
URLs	4,476	31.5	2,699	19.0	7,026	49.4	4,301	23.7	2,874	15.9	10,902	60.3	4,160	22.0	3,015	15.9	11,721	62.0
Forms / URL	0.14	-	0.13	-	0.11	-	0.15	-	0.12	-	0.07	-	0.15	-	0.11	-	0.08	-

TABLE III: Aggregated attack surface coverage (absolute and relative). A is YURASCANNER (not shown) and B the other tool.

App	Metric	Black Widow			BFS			Rand. BFS		
		$\setminus B$	$\cap$	$B \setminus$	$\setminus B$	$\cap$	$B \setminus$	$\setminus B$	$\cap$	$B \setminus$
Dolibarr	Forms	28	39	37	39	28	60	37	30	104
	URL	589	219	1,119	635	173	2,776	616	192	2,128
GitLab	Forms	20	43	60	28	35	63	25	38	73
	URL	116	317	334	107	326	218	97	336	260
LimeSurvey	Forms	14	30	36	21	23	9	36	8	2
	URL	403	227	374	420	210	82	520	110	15
MediaWiki	Forms	25	11	77	19	17	70	22	14	96
	URL	284	251	1,481	284	251	998	273	262	2,292
Moodle	Forms	132	123	101	112	143	273	119	136	207
	URL	1,229	643	2,040	1,072	800	5,723	1,115	757	4,879
OpenCart	Forms	82	1	184	78	5	52	71	12	72
	URL	394	90	217	378	106	399	266	218	901
Redacted	Forms	117	15	101	122	10	31	107	25	108
	URL	346	260	256	373	233	170	199	407	476
OwnCloud	Forms	2	10	52	2	10	29	1	11	47
	URL	33	125	134	27	131	148	25	133	161
phpBB	Forms	168	12	114	164	16	102	163	17	119
	URL	796	378	1,017	718	456	315	765	409	489
RedMine	Forms	44	55	29	50	49	50	50	49	80
	URL	286	189	54	287	188	73	284	191	120

TABLE IV: Per-application attack surface coverage. A is YURASCANNER (not shown) and B the other tool.

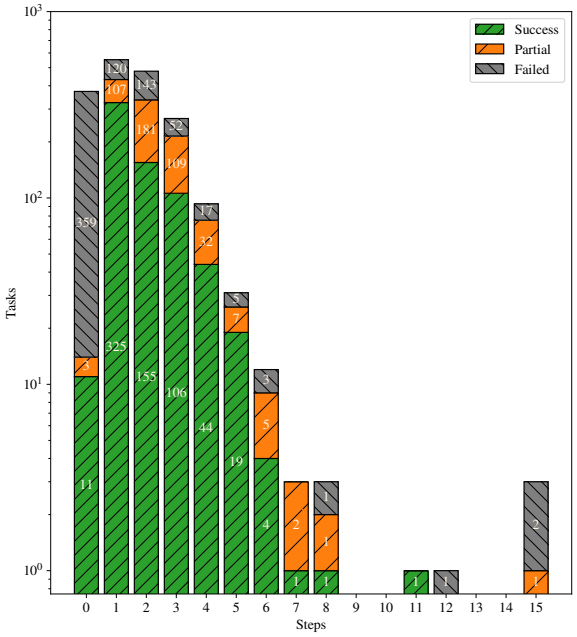

Fig. 7: Number of tasks by their length measured in the number of steps.

E. RQ4: Characterization of the New Attack Surface

The fact that YURASCANNER discovers fewer URLs and forms than the other tools may lead to wrong conclusions about performances. This result is a direct consequence of the unfocused crawling strategy implemented by the other tools: all clickables are equivalent and equally probable to be the next action. Such a strategy brings the crawler to diverse areas of a web application, increasing the chances of finding new URLs and forms. A strategy like the one of YURASCANNER keeps the crawler focused on achieving the task objective, reducing the chances of finding a more diverse set of URLs and forms. However, the major differentiating element between the two strategies is that the new attack surface discovered by YURASCANNER is far deeper than the one discovered by the other tools, showing the unique skills of YURASCANNER to handle complex application workflows.

Methodology — In this section, we take a look at the portion of the attack surface discovered by YURASCANNER, focusing on how our coverage increases as YURASCANNER executes tasks' steps.

Results — Figure 7 shows the distribution of the number of tasks over the number of steps that YURASCANNER executed. For failed tasks, we report the number of steps executed before the error. For partially and fully executed tasks, the number of steps corresponds to the task length. The majority of tasks covered a few steps. Zero-step tasks are predominantly failed executions due to a STOP command. Only 14 of these tasks were executed partially or completely. These were tasks such as "Browse the product catalog" where the catalog was shown in the seed URL. The longest tasks have 15 steps, of which two executions were partial and one was successful.

During the execution of these tasks, YURASCANNER collected forms from the visited pages. When examining the collection of forms in relation to the steps per task (Figure 8), we observe that 44% of the forms were also discovered by at least one of the other tools. These forms are located in the shallow areas of the web applications, at most at depth three. At the same depth, YURASCANNER discovered twice as many forms, i.e., 85.7%. The remaining 14.3% of the forms discovered by YURASCANNER are at a depth greater than three. All these forms appear to be out of reach for existing tools.

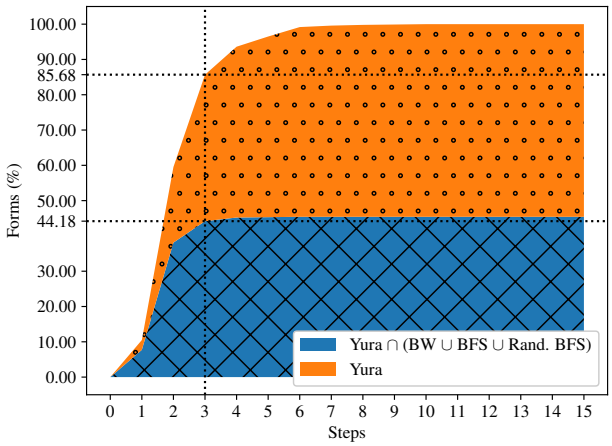

Fig. 8: Cumulative ratio of forms collected at each step.

Name	Tot.	Uniq.	Yura		BW	
			Stored	Reflected	Stored	Reflected
Redacted	12	11	4	7	-	1
GitLab	-	-	-	-	-	-
OpenCart	-	-	-	-	-	-
Redmine	-	-	-	-	-	-
Dolibarr	-	-	-	-	-	-
Moodle	2	1	1	-	1	-
MediaWiki	-	-	-	-	-	-
OwnCloud	-	-	-	-	-	-
LimeSurvey	-	-	-	-	-	-
phpBB	-	-	-	-	-	-
NextCloud	-	-	-	-	-	-
Joomla	-	-	-	-	-	-
Leantime	1	1	-	-	1	-
EspoCRM	-	-	-	-	-	-
iTop	-	-	-	-	-	-
MintHCM	-	-	-	-	-	-
Mautic	-	-	-	-	-	-
GLPI	-	-	-	-	-	-
Silverpeas	-	-	-	-	-	-
Monica	-	-	-	-	-	-

TABLE V: Discovered vulnerabilities.

F. RQ5: Vulnerability Detection

Methodology — We now look at the detection of XSS vulnerabilities in all 20 web applications of the TE+VD dataset. For this experiment, we compared our tool with Black Widow only. Both tools were let run for four hours each.

We pointed the tools to the admin section of a website to find reflected cross-site scripting vulnerabilities that unauthenticated users could exploit by sending a vulnerable link to an admin user.

Results — In total, YURASCANNER and Black Widow reported 15 XSS vulnerabilities across three web applications: *Redacted* (YURASCANNER 11 and Black Widow one), Moodle (YURASCANNER one and Black Widow one), and Leantime (Black Widow one). We manually reviewed all reports to confirm the presence of the vulnerabilities. All identified vulnerabilities were true positives, and we further classified them as either stored or reflected XSS, as shown in Table V. Only the vulnerability found in Moodle is not a direct XSS vulnerability but an HTML meta tag attribute injection vulnerability, which could escalate into an XSS vulnerability.

Out of the 15 reports, four referred to two unique vulnerabilities, resulting in a total of 13 unique vulnerabilities. All these vulnerabilities are zero-day, with 12 discovered by YURASCANNER. These vulnerabilities are located between four and two clicks away from the main page: three vulnerabilities are four clicks away from the starting page, two vulnerabilities are three clicks away, and the remaining six, including the one found by Black Widow, are no deeper than two clicks away.

The vast majority of the discovered vulnerabilities can be exploited by non-admin users. Seven are reflected XSS that a non-authenticated user can exploit, and one is a stored XSS that a low-privileged user can exploit. We could not find an exploit for non-admin users for the remaining five stored XSS. We discuss the vulnerability notification in Section VI-B.

VI. DISCUSSION

A. Results

In this section, we discuss the main results of this paper.

1) A New Approach and New Opportunities to Explore: This paper presents one of the first task-driven scanning approaches, proposing an initial concrete solution to the problem of defining, generating, and executing tasks on web applications. This approach serves as a stepping stone to improve attack surface coverage, particularly for areas that are currently difficult to reach or completely unexplored.

Task Generation — Our experiments show that YURASCANNER can successfully generate tasks with the aid of an LLM. 77% of them are tasks associated with valid functionalities offered by the web applications. The invalid tasks were mostly caused by insufficient context description, resulting in incoherent and inconsistent answers (i.e., hallucinations). For example, if the page representation consists of a single *Login* button, LLM could reply with a list of tasks that are irrelevant to the application under test.

Task Execution — YURASCANNER is effective at executing tasks, managing to execute correctly or partially 61.3% of the valid tasks. The remaining errors are a combination of direct failure, e.g., deviations and no actions, or indirect, a state missing because of a previous error (direct).

Execution Accuracy and Tasks Dependency — This work has highlighted non-trivial dependencies between the execution accuracy and the dependency between tasks. First, if a state-changing task fails (e.g., adding an item to the database), then all other tasks relying on that task (e.g., viewing or modifying that item) will also fail. This is evident from the spike in missed state errors likely caused by deviations and no action errors. Second, task generation can produce tasks that block other tasks. For example, when testing Leantime, an application from the VD dataset, a task was to "Delete a user from the management section," which YURASCANNER

executed successfully. Unfortunately, that was the only user account in the system, locking YURASCANNER out and causing all subsequent tasks to fail.

2) Task-driven Complements Traditional Crawling: The main result of our paper is demonstrating that task-driven crawling complements traditional crawling. Our results indicate that each approach excels in covering different areas of the attack surface with distinct properties: traditional scanning excels in exploring shallow areas, whereas a task-driven approach is more effective for deeper areas.

New Attack Surface — YURASCANNER discovered about +55.19% more forms and +35% more URLs when compared with the other tools.

New Attack Surface is Deeper — The attack surface discovered by our and the other tools is orthogonal, where only a small fraction of the surface is discovered by the two. Such a common surface contains forms that are in shallow areas of the web application (at most three steps away from the seed URL). The unique surface discovered by YURASCANNER is deeper, with new forms discovered up to 15 steps deep.

New Vulnerabilities in the Deeper States — This paper targeted real-size, modern web applications, used by millions of users. These applications showed a higher resilience towards the payloads used by Black Widow. Nevertheless, our task-driven approach was able to find 11 vulnerabilities in *Redacted* (four stored XSS and seven reflected XSS). A closer look at the vulnerabilities that we discovered, reveals that they are located from four up to two steps away from the seed URL, showing the ability of YURASCANNER to identify forms of long workflows.

B. Ethical Discussion

Experiments Overhead — The experiments described in this paper were conducted on web applications installed on our servers. The overhead caused by our experiments, such as additional network traffic and CPU and memory workload, was sustained by the internal infrastructure of our institution and did not affect the developers or users.

Vulnerability Notification — Our experiments identified previously unknown vulnerabilities in *Redacted*, Moodle, and Leantime. We have established communication channels with the developers and reported the vulnerabilities. Leantime developers have fixed the vulnerability in version 3.3.0. Moodle developers have replied that the vulnerability has no impact under the application's threat model. *Redacted* developers have acknowledged the presence of vulnerabilities and their impact but did not provide a timeline for the fix. Therefore, we anonymized the name of their application in the paper.

Open Science and Ethical Considerations — The authors of this paper are deeply committed to an open science policy. We will share the artifacts of this paper, including the Docker files of the web applications, the configuration, and other artifacts generated and collected during the execution of the experiments. However, we acknowledge that a web scanner capable of executing tasks may pose new risks that prior research did not. In particular, YURASCANNER may be misused to achieve other goals than the ones intended in this paper, exacerbating issues such as fake account creations and scraping. Accordingly, we will not publicly share the source code of YURASCANNER. Instead, we have set up a submission form[3] where interested researchers can apply, stating their objectives. We will vet the provided information and grant or deny access to the source code.

VII. RELATED WORK

Traditional Techniques — Traditional web scanning techniques have been a cornerstone of web application security for many years. Early scanners like w3af [14], Skipfish [23], Wapiti [25] and ZAP [13] set the foundation by performing common checks for known vulnerabilities such SQLi [26], file inclusion [27], and XSS [28], [29], often testing a dictionary of known exploits. However, these scanners were primarily designed to scan static pages and lacked the sophistication needed to handle the dynamic nature of web applications such as client-side and server-side states. To address these shortcomings, multiple scanners have been proposed by security professionals (e.g., Wapiti [25] and Arachni [12]), academia (e.g., Enemy of the State [5] and jÄk [2]), and industry (e.g., Burp [30]). For example, Enemy of the State scanner [5] utilizes a heuristic-based approach to deduce the server's state by analyzing how different requests lead to variations in the links present on web pages. However, reliance on heuristic methods often leads to incomplete or inaccurate state inferences. Other crawlers like Skipfish [23], ZAP [13], Arachni [12] and w3af [14] prioritize performance over server-side state by using parallel requests. In comparison, jÄk [2] focuses on the client-side state, capturing JavaScript events to enhance coverage, but jÄk's event handling is limited, e.g., it is unable to capture form submission events. Scanners like Enemy of the State, w3af, and ZAP can handle JavaScript-based UIs, but they do not model events, thereby failing to detect vulnerabilities that depend on them (e.g., form submissions, clicks, etc).

State-of-the-art Scanners — A more recent line of work combined dictionary-based scanning with in-browser dynamic testing and dynamic taint tracking techniques [3], [31], [32]. For example, Black Widow [3] is one of such approaches. It creates a navigation model of the application by triggering static structures in the UI (e.g., anchors, forms, etc) and JavaScript events (e.g., mouse clicks), enabling it to retrace its steps. Then, during the page visit, it enriches the model with data flow information using dynamic testing. It identifies input fields, probes them with test values, and then monitors their reappearance in the HTML document (i.e., reflected XSS). Black Ostrich [33] extends Black Widow by improving the crawler's ability to pass validation checks on input fields. It incorporates string constraint solving to dynamically infer valid inputs from regular expression patterns in web applications. Other works [31], [32] scanned websites for vulnerabilities using a fully-fledged dynamic taint tracking approach.

Scanning Protected States — A different line of work proposed scanners that can reach pages after the login. The

[3] https://github.com/pixelindigo/yurascanner/tree/ndss25

first category of these scanners is those based on a manually curated list of login scripts (e.g., [34], [35]). However, such approaches do not scale due to manual analysis requirements. The second category scanners leverage SSOs for automated login (e.g., [36]). However, these scanners often struggle with the diverse and evolving SSO protocols, requiring frequent updates and limiting their effectiveness to only applications using SSOs, leaving out many that use custom authentication mechanisms. The third category consists of the scanners relying on pattern matching and regular expressions, e.g., Cookie Hunter [37] and Shepherd [38]. However, these approaches are too brittle to minor changes in the UIs, hampering their effectiveness. For instance, Cookie Hunter has a high failure rate in automated sign-ups, with 88% of attempts failing. Additionally, they require continuous creation and maintenance of new patterns, which also does not scale. In comparison, our approach can handle compound client-server states, like multi-step shopping processes, allowing it to follow complex workflows to reach deeper areas for security scans.

Using LLMs for Web Interactions — Recently, we have seen new ideas using LLMs to visit pages. Deng et al. [39] introduced Mind2Web, a dataset for evaluating web agents that can follow language instructions to complete tasks on websites. As part of the same work, they implemented MindAct, an agent that uses language models to predict the next action on a webpage given a task. Deng et al. proposed PentestGPT [40], a semi-automated and interactive penetration testing tool that relies on LLM to guide testers in order to streamline the penetration testing process. For instance, given an IP address as input for testing, it outputs step-by-step guidance for various tests (e.g., nmap scan, identifying open ports, enumerating FTP services, etc). Finally, Fang et. al. [41] explored whether LLM agents can be abused for autonomous website hacking. Their work explores the potential for malicious applications of LLMs, while ours focuses on leveraging their capabilities for ethical security testing, increasing the coverage by reaching deep application states.

Other non-academic works have demonstrated how LLMs can be utilized to scrape content from webpages [42], [43], while projects such as Natbot [9] and Skyvern [44] assist users in navigating pages.

Out of the tools mentioned above, MindAct, Natbot, and Skyvern share the most similarities to YURASCANNER since they aim to execute diverse tasks in web applications using LLMs. However, their main objective is to assist users in performing manually provided tasks. In contrast, YURASCANNER targets web application coverage and does not rely on user-provided tasks. Instead, YURASCANNER extracts the tasks automatically and executes them to explore the web application without human guidance.

Additionally, the most significant difference is how YURASCANNER handles form submissions. Natbot, MindAct, and Skyvern fill out forms using the same prompt they use for navigation. Moreover, in the case of Natbot and MindAct, each navigation action can only fill out a single input field. In contrast, YURASCANNER uses a separate prompt to fill a form in one go.

VIII. CONCLUSION

This paper presents one of the first task-driven scanning approaches and implementation (YURASCANNER), providing a foundational solution to the challenge of defining, generating, and executing tasks on web applications. Our methodology leverages large-language models to autonomously navigate complex workflows, offering significant improvements in attack surface coverage, particularly in areas that traditional scanners fail to reach. By modeling the scanner as a goal-based agent and integrating an XSS engine, we demonstrated a comprehensive system capable of uncovering deeper states and new vulnerabilities within web applications.

The primary outcome of this paper is the demonstration that task-driven crawling effectively complements traditional scanning techniques. Our experiments reveal that while traditional scanning methods are adept at exploring shallow areas of the attack surface, the task-driven approach excels in delving into deeper, more intricate regions. This complementary nature results in a more thorough exploration of web applications, as evidenced by YURASCANNER discovering a substantial number of new attack surfaces and zero-day XSS vulnerabilities. These findings underscore the potential of task-driven scanning to enhance web security by addressing previously unexplored areas and improving overall vulnerability detection.

ACKNOWLEDGMENT

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 452850842.

REFERENCES

[1] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling AJAX by Inferring User Interface State Changes," in *2008 Eighth International Conference on Web Engineering*, 2008, pp. 122–134.

[2] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications," in *Research in Attacks, Intrusions, and Defenses*, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 295–316.

[3] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black Widow: Blackbox Data-driven Web Scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142.

[4] A. Stafeev and G. Pellegrino, "SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 719–737. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/stafeev

[5] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[6] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic Web Testing Using Curiosity-Driven Reinforcement Learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 423–435.

[7] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging User Interactions for In-Depth Testing of Web Applications," in *Recent Advances in Intrusion Detection*, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 191–210.

[8] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang, "Reinforcement Learning on Web Interfaces using Workflow-Guided Exploration," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=ryTp3f-0-

[9] N. Friedman. (2022) Natbot. https://github.com/nat/natbot.

[10] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, 2016.

[11] A. Doupé, M. Cova, and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010.

[12] T. Laskos. Arachni. [Online]. Available: https://www.arachni-scanner.com/

[13] (2010) OWASP Zed Attack Proxy. https://www.zaproxy.org/.

[14] A. Riancho. (2007) w3af: Open Source Web Application Security Scanner. http://w3af.org/.

[15] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[16] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative Agents: Interactive Simulacra of Human Behavior," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606763

[17] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging LLM-as-a-judge with MT-bench and Chatbot Arena," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[18] AI@Meta. (2024) Llama 3. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[19] G. Team and G. DeepMind, "Gemma 2: Improving Open Language Models at a Practical Size," 2024. [Online]. Available: https://storage.googleapis.com/deepmind-media/gemma/gemma-2-report.pdf

[20] Bitnami Application Catalog. Last accessed June 2024. [Online]. Available: https://bitnami.com/stacks

[21] Elestio. Last accessed June 2024. [Online]. Available: https://elest.io/fully-managed-services/applications

[22] DockerHub. Last accessed June 2024. [Online]. Available: https://hub.docker.com/search?categories=Content+Management+System

[23] M. Zalewski. (2010) Skipfish. https://code.google.com/archive/p/skipfish.

[24] S. Hykes. Docker. [Online]. Available: https://www.docker.com/

[25] N. Surribas. Wapiti. [Online]. Available: https://github.com/wapiti-scanner/wapiti

[26] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql injection attacks and countermeasures." in *ISSSE*, 2006.

[27] Y. Makino and V. Klyuev, "Evaluation of web vulnerability scanners," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2015, pp. 399–402.

[28] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of DOM-based XSS," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1193–1204. [Online]. Available: https://doi.org/10.1145/2508859.2516703

[29] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 736–747. [Online]. Available: https://doi.org/10.1145/2382196.2382274

[30] BurpSuite. Last accessed June 2024. [Online]. Available: https://portswigger.net/burp

[31] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 236–250.

[32] S. Khodayari, T. Barber, and G. Pellegrino, "The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 166–184.

[33] B. Eriksson, A. Stjerna, R. De Masellis, P. Rüemmer, and A. Sabelfeld, "Black Ostrich: Web Application Scanning with String Solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 549–563.

[34] J. Rautenstrauch, M. Mitkov, T. Helbrecht, L. Hetterich, and B. Stock, "To Auth or Not To Auth? A Comparative Analysis of the Pre- and Post-Login Security Landscape," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[35] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2020.

[36] Y. Zhou and D. Evans, "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 495–510. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou

[37] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1953–1970.

[38] H. Jonker, S. Karsch, B. Krumnow, and M. Sleegers, "Shepherd: A Generic Approach to Automating Website Login," in *Proceedings of the 2020 Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, United States, 2020.

[39] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su, "Mind2Web: towards a generalist agent for the web," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[40] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 847–864. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/deng

[41] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang, "LLM Agents can Autonomously Hack Websites," 2024. [Online]. Available: https://arxiv.org/abs/2402.06664

[42] T. Connors. (2024) Building a Universal AI Scraper. https://timconnors.co/posts/ai-scraper.

[43] (2023) CrawlGPT. https://github.com/gh18l/CrawlGPT.

[44] (2024) Skyvern. https://github.com/Skyvern-AI/skyvern.

APPENDIX A
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: We have set up the vetting process to ensure that YURASCANNER would not be misused. Up-to-date instructions for source code access are located at the following link: https://github.com/pixelindigo/yurascanner/tree/ndss25.

2) Hardware dependencies: We tested the setup on a four core system with 16GB of RAM and 60GB of disk space. This should be sufficient for running single experiments. Running more experiments in parallel might require a higher RAM capacity and core count.

3) Software dependencies: We tested the setup on Ubuntu 22.04.4 with Docker version 24.0.7, docker-compose version 1.29.2 and Python 3.10.12.

4) Benchmarks: We use the Arachnarium[1] framework to run the experiments.

B. Artifact Installation & Configuration

Install required packages

```
sudo apt update && sudo apt
install -y docker.io docker-compose
python3-virtualenv
```

Add your user to docker group

```
sudo usermod -aG docker user
```

Enable docker service

```
sudo systemctl enable docker
```

Reboot your system and ensure that your user is in docker group

`id` command output should contain `docker`

Create and activate a virtualenv

```
virtualenv .env && source
.env/bin/activate
```

Install required Python packages

```
pip install -r requirements.txt
```

C. Experiment Workflow

The artifact is comprised of two parts: a) raw experiment data with the scripts to generate the figures and b) the YURASCANNER source code and the necessary Arachnarium modules to run the experiments.

D. Major Claims

- (C1): YURASCANNER can automatically generate and execute tasks on a website. This is proven by the experiment (E1) whose results are reported in Section V.C.
- (C2): YURASCANNER demonstrates orthogonal capabilities, when compared to the other techniques, potentially covering areas of the attack surface with different inherent characteristics. This is proven by the experiment (E2) whose results are reported in Tables III and IV.
- (C3): The new attack surface discovered by YURASCANNER is far deeper than the one discovered by the other tools, showing the unique skills of YURASCANNER to handle complex application workflows. This is proven by the experiment (E3) whose results are illustrated in Figures 7 and 8.
- (C4): YURASCANNER has been used to uncover zero day bugs in *Redacted* and Moodle. This is proven by the experiment (E4) in Section V.F.

E. Evaluation

The scale of the experiments in the paper is quite large (10-20 webapps $\times$ 2-4 scanners $\times$ 4 hours $\approx$ 80-320 compute hours). Typically, the compute hours could be scaled down significantly if any form of parallelization is employed. Unfortunately, we also rely on OpenAI API which has organization-wide rate limits. As there are multiple research groups in our organization, we limit the YURASCANNER experiments to a single experiment at a time. Therefore, we suggest running scaled-down experiments, i.e., running a single web application instead of all twenty.

1) Experiment (E1): [Task Execution Classification] [60 human-minutes + 4 compute hours]: run the experiment with *Redacted* web application and manually verify the task execution capabilities of YURASCANNER.

[How to] Run the experiment with YURASCANNER and observe it's task execution capabilities.

[Execution]

Run the *Redacted* experiment

```
arachnarium run crawlers/yurascanner
apps/redacted http://web/redacted/admin/
--screenshot --username admin --password
password --gpt4 --autotask --headless -t
240
```

Navigate to the screenshots folder

```
cd experiments/redacted/yurascanner/
```

```
cd <uuid>/report/screenshots/<timestamp>
```

Observe the generated jpg files

[Results] Observe the screenshots of the run. The bottom of a screenshot would display the current task being executed. Note that not all tasks would be executed successfully, as showed in Table II. Instead, some of the tasks would be executed partially (reaching the functionality but failing to execute the last action) or even deviate from the target functionality.

2) Experiment (E2): [Attack Surface Coverage] [10 human-minutes + 10 compute minutes]: extract the URLs and the forms discovered during the experiment, compare them against the ones discovered by the other tools and show the results in the tables.

[How to] The csv files for tables III and IV are generated in `experiment_data/scripts/tables.py` as a part of `generate_figures.sh` script.

[Preparation] We will be using the raw results of `task_execution` experiment to generate the tables. Alternatively, populate `task_execution` folder with the contents of `experiments` folder.

[Execution]

[1] https://github.com/pixelindigo/arachnarium

Run the script to generate the figures

```
bash generate_figures.sh
```

[Results] The `figures/` folder should contain `table4_full_attack_surface.csv` and `table3_attack_surface.csv` files, which would have a similar structure to the tables III and IV in the paper.

3) Experiment (E3): [Characterization of the New Attack Surface] [10 human-minutes + 10 compute minutes]: extract the forms discovered during the experiment and the length of the tasks, compare them against the ones discovered by the other tools and show the results in the figures.

[How to] The figures 7 and 8 are generated in `experiment_data/scripts/plot_steps.py` and `experiment_data/scripts/form_analysis.py` as a part of `generate_figures.sh` script.

[Preparation] We will be using the raw results of `task_execution` experiment to generate the tables. Alternatively, populate `task_execution` folder with the contents of `experiments` folder.

[Execution]

Run the script

```
bash generate_figures.sh
```

[Results] The `figures/` folder should contain `figure7_steps.pdf` and `figure8_forms_cumulative.pdf` files, which should show a similar figure to the figures 7 and 8 in the paper.

4) Experiment (E4): [Vulnerability Detection] [10 human-minutes + 8 compute hours]: run the experiment with *Redacted* and Moodle web applications and show that YURASCANNER finds zero-day bugs.

[How to] The raw data already contains the logs of the found bugs. Otherwise, run a new experiment and check the results.

[Preparation] If you want to use the raw data collected during our experiments leave the files as is and run `python scripts/get_yura_vulns.py` and `python scripts/get_bw_vulns.py` from within the `experiment_data` folder.

Otherwise, run the experiment similarly to (E1). Replace the `vulnerability_detection` folder with the contents of `experiments` folder.

[Execution]

[Optional] Run the *Redacted* experiment

```
arachnarium run crawlers/yurascanner
apps/redacted http://web/redacted/admin
--screenshot --username admin --password
password --gpt4 --autotask --headless -t
240
```

[Optional] Run the Black Widow experiment

```
arachnarium run crawlers/blackwidow
apps/redacted --url http://web/redacted/admin
-t 240
```

Run the scripts

```
python scripts/get_yura_vulns.py
vulnerability_detection | uniq

python scripts/get_bw_vulns.py
vulnerability_detection | uniq
```

Or (If you run the experiments yourself)

```
python scripts/get_yura_vulns.py
experiments | uniq

python scripts/get_bw_vulns.py
experiments | uniq
```

[Results] Observe the result of `get_*_vulns.py` scripts as it would contain the urls on which the vulnerabilities were found. Note that due to the randomness (Black Widow uses a randomized navigation, OpenAI models are not static) the results might be different to ours.

F. Notes

For ethical reasons, which are listed in the paper in Section VI.B, we will not be making the source code of YURASCANNER publicly available. Instead, we would provide the source code on request. The process is described in Section VI.B.