SCWAD: Automated Pentesting of Web Applications

Natan Talon[4], Valérie Viet Triem Tong[1], Gilles Guette[3], Yufei Han[2] and Youssef Laarouchi[4]

[1] *CentraleSupélec, Rennes, France*

[2] *Inria, Rennes, France*

[3] *Université de Rennes, Rennes, France*

[4] *Hackuity, Lyon, France*

Keywords: Pentest Automation, Web Application.

Abstract: A wide array of techniques and tools can be employed for web application security assessment. Some methods, such as fuzzers and scanners, are partially or fully automated, offering speed and cost-effectiveness. However, these tools often fall short in detecting specific vulnerabilities like broken access control and are prone to generating false positives. On the other hand, manual processes like penetration testing, though more time-consuming and necessitating expertise, provide a more comprehensive risk assessment. To overcome the limitations of automated tools, these techniques are frequently combined. Fuzzers and scanners, despite their ease of use and quick results, require the expertise of penetration testing experts to address their limitations. By integrating these approaches, a more robust and nuanced security assessment can be achieved. This article presents SCWAD, an automated and customizable penetration testing framework designed to assess vulnerabilities in web applications.

1 INTRODUCTION

Penetration testing (pentest) is an audit technique employed to assess the security risk of an information system, conducted by skilled security experts (pentesters). These pentesters not only identify vulnerabilities within the targeted system but also exploit these vulnerabilities to illustrate their potential impact when woven into an attack scenario. In the realm of *web application* pentesting, pentesters face a myriad of frameworks, architectures, and programming languages that underpin these applications.

Effectively organizing web application pentesting poses significant challenges, primarily twofold. *First* web application faces to multiple type of vulnerabilities *Second*, a manually organized web application pentest is a costly and time-consuming undertaking.While automated tools like fuzzers and scanners can assist in vulnerability assessments, their capability is limited to a fraction of vulnerabilities

In response to existing limitations, our study presents SCWAD as an automated pentesting framework tailored for effective evaluation of web applications. The fundamental concept behind SCWAD lies in framing automated web pentesting as a sequential decision-making process.

The major contribution in our study can be summarized in the following perspectives.

- We conceptualize pentesting web applications as a sequential decision-making challenge and introduce SCWAD as an automated framework for web application pentesting.
- We structure the information acquired during a pentest campaign in a knowledge base. We propose a method for highlighting the vulnerabilities by querying this base.
- We organise a comparative study between SCWAD and the state-of-the-art practices of fuzzers and scanners for web applications, including Portswigger's BurpSuitePro scanner (PortSwigger, 2023), OWASP's ZAP scanner (OWASP, 2023b) and Wapiti's fuzzer (Surribas, 2023).
- We demonstrate that SCWAD is capable of handling large modern applications. At the same time, we point out that these applications are too large to be manually pentested.

In the followings, Section 2 surveys the works in pentesting tools related to our work. Section 3 de-

Talon, N., Tong, V., Guette, G., Han, Y. and Laarouchi, Y.
SCWAD: Automated Pentesting of Web Applications.
DOI: 10.5220/0012721000003767
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 424-433
ISBN: 978-989-758-709-2; ISSN: 2184-7711
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

tails the different modules contituting SCWAD. We further present how a pentesting task is organised in an iterative way with the three modules of SCWAD. Section 4, instantiates the exploration process of three vulnerabilities on web applications, which demonstrates the use of SCWAD for automated pentesting. Then it compares SCWAD and three popular fuzzer / scanner tools on dedicated web application and challenges SCWAD on online applications.

2 BACKGROUND

2.1 Web Applications and Their Associated Risks

The term *web application* denotes application software accessible through the web and executed via a web browser. A web application can be a straightforward static website where content is delivered exactly as stored on the server for each user. In contrast, dynamic web applications are tailored to individual users and rely on a three-tier architecture, separating presentation, application processing, and data management functions. Web applications find utility across various domains, from basic email clients to complex e-commerce platforms and online games. Their widespread adoption is due to their user-friendly interface and global accessibility, allowing users to engage from anywhere. However, this accessibility also renders web applications vulnerable, making them prime targets for attacks, ranging from simple denial-of-service to more sophisticated campaigns. Attacks on web applications can grant attackers unauthorized access, potentially compromising the entire system. Furthermore, web applications may serve as vectors for information leakage.

In 2021, the Open Web Application Security Project (OWASP, 2021) released its updated top 10 risks associated with designing or implementing web applications. Topping the list is the vulnerability of `Broken Access Control`, a flaw enabling users to act beyond their designated permissions without prior authentication. Exploiting this flaw could result in information disclosure, modification, or destruction of essential business functions. Another critical vulnerability, ranking third, is 'Injection,' allowing attackers to manipulate the system's interpretation of commands. One such instance is the widely known Reflected Cross-Site Scripting (XSS) attacks, occurring when malicious code is sent to a victim end user through a web application.

During a reflected XSS attack, victims are deceived into executing a malicious payload, often embedded in a link or a crafted form. The injected codes appear legitimate, camouflaging themselves as payloads from a trusted server, ultimately making the victim's web browser the target. These attacks exploit improper sanitization of user input within the web application, allowing the injected code to end up in the HTML output. Reflected server XSS exploits employ various techniques to bypass insufficient input sanitization checks, often leading to significant data leaks in web applications, as revealed by (Buyukkayhan et al., 2020).

2.2 Vulnerability Hunting in Web Applications

To bolster the security of web applications, a systematic search for vulnerabilities is often conducted. To conduct this search, the literature (Zhang et al., 2022) provides solutions of two types. First static analysis that focuses on analyzing the source code of the web application. This method offers the advantage of scrutinizing code without executing programs. Numerous tools have been developed to address PHP scripts and taint-based vulnerabilities, such as Pixy (Jovanovic et al., 2006) and its object-oriented extension, OOPixy (Nashaat et al., 2017), phpSAFE (Nunes et al., 2015), and RIPS (Dahse and Holz, 2014). Pixy, for instance, employs a flow-sensitive, interprocedural, and context-sensitive data flow analysis to uncover vulnerabilities like SQL injection, cross-site scripting, and command injection, albeit with a false positive rate of around 50%. phpSAFE serves as a source code analyzer for PHP-based plugins, capable of detecting Cross Site Scripting and SQL Injection vulnerabilities. RIPS, on the other hand, utilizes the abstract syntax tree of PHP scripts and employs backward-directed taint analysis to identify taint-based vulnerabilities. In a benchmark study by (Nunes et al., 2018), which featured 134 WordPress plugins with real vulnerabilities, Pixy and other static analysis tools were assessed for XSS and SQLi vulnerabilities. The results indicated that there is not a one-size-fits-all tool for all scenarios and classes of vulnerabilities.

Static analysis is inherently limited to a single programming language, typically PHP or JavaScript in the tools available today. This restriction means that vulnerabilities in source codes written in other programming languages go undetected by the static analysis method. Furthermore, static analysis cannot reveal values or content that only become apparent through the dynamic analysis of the target web application, such as cookie values.

The second type of solution is dynamic analysis that focuses on discovering vulnerabilities while the web application is running and observes the application's output behavior in response to specific inputs. These analyzers are often referred to as 'black-box scanners' (Kals et al., ; Drakonakis et al., 2023; Eriksson et al., 2021; Pellegrino et al., 2015) because they assume that the application's internals are not observable. Traditional black-box scanners aim to enumerate all reachable pages and then fuzz input data, including URL parameters, form values, and cookies, to provoke vulnerabilities. In their evaluation of 11 black-box web vulnerability scanners, (Doupé et al., 2010) emphasized the importance of deep crawling to discover all vulnerabilities in an application. However, as noted in (Doupé et al., 2012), these scanners *ignore a key aspect of modern web applications: any request can change the state of the web application*. Doupé et al's work is credited with pioneering the use of a state machine to guide state-aware fuzzing of web applications, resulting in improved code coverage compared to traditional scanners.

Finally, the security of web applications can be assessed through pentest campaigns, which simulate attacks to determine system security. Web pentesting is typically conducted by human experts who manually explore the application to identify vulnerabilities, often aided by one or more web application scanners. In this context, measuring test coverage and reproducing the test campaign can be challenging. In our study, we echo to this challenge by proposing SCWAD as an automated and customizable vulnerability exploration tool. Security analyst/service owner of web applications can use SCWAD to provide a comprehensive coverage of possible vulnerabilities, in order to achieve an accurate vulnerability assessment. In the next section, we describe our pentest's modelisation.

3 SCWAD FRAMEWORK

We introduce SCWAD an autonomous pentesting framework to spotlight vulnerabilities within web applications. SCWAD employs an attack-based assessment strategy, simulating the role of an attacker. It systematically explores potential vulnerabilities in a given web application, exploiting identified weaknesses to trigger data breaches. Moreover, SCWAD is equipped with the capability to register and replay pentest campaigns.

SCWAD operates through three key components that interact in a continuous loop.

Update knowledge base encompasses defining and updating of a **Knowledge Base** specific to each web application tested within SCWAD. This knowledge base encapsulates insights gathered by a pentester across multiple sessions, assuming various user roles (see Section 3.1). *Act on the app* involves an automated **Agent** responsible for devising and executing pertinent actions on the web application. This agent's primary objective is to comprehensively explore the application, attempting simulated attacks, and enriching the knowledge base as extensively as possible (see Section 3.2). *Check for new vulnerabilities* is defined by introducing a **Verifier**, which is tasked with confirming the presence of vulnerabilities within a web application (see Section 3.4). SCWAD's Agent and Verifier may require expert knowledge to either build relevant actions or look for specific events while checking vulnerabilities. These expert knowledge are provided on query by the last component of SCWAD, the **Oracle** (see Section 3.2). The SCWAD process unfolds in structured rounds, with successive calls to the Agent, Knowledge Base update, and Verifier, as illustrated in Figure 1.

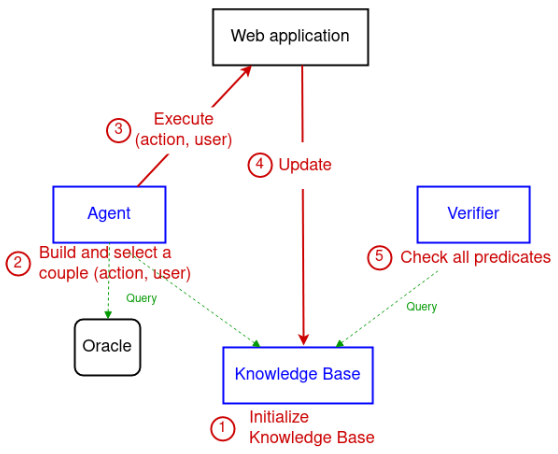

Figure 1: Components of SCWAD and their interaction.

3.1 SCWAD Knowledge Base

In SCWAD, the knowledge base of a web application stores in a structured format all information gathered by SCWAD about the application.

A web application is represented by an IP address or a domain name, enabling the identification of the web application's location and by the collection of information obtained while acting in various user roles within the application. Each user **u** is further described by:

u.login: Its user name

u.credentials. Its credentials for the application.

u.cookies. Its set of cookies, separated into active or not.

u.pages. The pages already visited by the user and those to which he/she knows a link. The record

pages provides details about the content of all of these pages. A page is modeled in the database though the web application's path, the HTTP methods, headers, and parameters used to access the page and the code of the page, offering information about its interactive elements such as clickable buttons, HTML forms, and more. A page must have its path defined. However, its access method and its can be unknown if the page has not been visited yet; in such cases, their values are set to none.
u.current_page: the current page visited by the user.
u.allowed_paths. all the links to the pages reachable by the user. These links are those found on the pages as the application is explored under the user's account. It includes anchors' references, images and scripts's sources and forms actions.

3.2 SCWAD Agent

The SCWAD agent is responsible for extending vulnerability exploration of web applications and enhancing the agent's knowledge base regarding the web applications to its fullest extent. To accomplish this objective, the SCWAD agent undertakes three key tasks: it compiles a list of potential actions via the knowledge base's current status. It selects an action from this list and executes it. The execution of the chosen action promptly initiates a knowledge base update and triggers a call to the SCWAD verifier, concluding the current round of exploration.
Set of Actions. The SCWAD agent supports a range of potential actions based on four generic commands: `AccessWebPage`, `SendHttpForm`, `SearchData`, and `SetCookie`. Details regarding these commands, their parameters, and anticipated outcomes can be found in Table 1. An action is constructed using a command and concrete values $v_1, \cdots, v_n$ representing elements like URLs, cookies, or specific form values. These values are extracted from either the knowledge base or expert values served by the *oracle*. SCWAD oracle is simply a string generator designed to highlight XSS vulnerabilities. It is largely inspired by the *XSS Filter Evasion Cheat Sheet* (OWASP, 2023a) provided by OWASP.

Finally the set of possible actions are determined by all commands and all the possible combinations of concrete values.
Pentesting Strategy. From the available actions applicable in the current knowledge base state, the SCWAD agent selects one to execute. This choice of action determines the crawling strategy and, consequently, the tool's performance in vulnerability identification. In our work, we introduce four strategies. *Random selection* where the agent picks an action at random. This strategy serves as baseline comparison for the three others. *Explore first* prioritizes actions leading to accesses to new web pages. This strategy tries to gather all possible navigation information before attempting any exploitation. *Fill forms first* prioritizes actions leading to filling to new forms. This strategy attempts to exploit injection points as soon as they are found. Finally, *vulnerability tracking* consists in following specific sequences of actions designed from human pentesters behavior.

3.3 Update Knowledge Base

SCWAD performs actions on the current page of a known user in the knowledge base. Execution of an action succeeds when the server commits to perform, on the contrary, an execution fails when the server refuses to perform the action.

In case the action succeeds, SCWAD verifies if the user changed (e.g. login action) and enrich the database with the new knowledge (pages, cookies) gathered by both potential users.

3.4 Checking Vulnerabilities

A pentest campaign conducted with SCWAD starts with a knowledge base denoted as K_0 and progresses to an updated knowledge base K_i after executing a sequence of actions $\mathsf{X} : a_0 \ldots a_{i-1}$. SCWAD's verifier diligently checks for new vulnerabilities each time there are changes in the knowledge base. In this paper, we propose to translate the vulnerabilities into Boolean formulas in first-order logic. When the evaluation of a such a formula holds 'true' on a knowledge base $\mathbf{K}_j$, we assert that the sequence of actions from K_0 to K_j has led to the discovery of a vulnerability, formally characterized by the the formula. In this study, we showcase how SCWAD evaluates potential security risks in web applications. We focus on exploring three vulnerability types as per (OWASP, 2021) standards: A01:2021-Broken Access Control (referred to as BAC in the rest of the paper), XSS injection, categorized under A03:2021-Injection, and technical information disclosure (referred to as TID), which falls under A05:2021-Security Misconfiguration.

A Broken Access Control (BAC) vulnerability appear when a user can access data or functionality of an other user when it should not. For example, in a shop web application a user should not access the cart of an other user. An reflected XSS vulnerability appear when an attacker can provide javascript code as inputs to the server instead of genuine data. This code must then be reflected on the client and executed.

Table 1: List of commands and corresponding parameters in SCWAD.

Command	Parameters	Expected behavior on the Web application
`AccessWebPage`	`u`: *url*	change current page for the current user
`SendHttpForm`	`x`: *xpath* , `input`: *dict* { *key: value* }	send the form identified by `x` and filled with `input` to the web application
`SearchData`	`data`: *string*	return `true` if `data` has been found in the current page
`SetCookie`	`cookie`: *dict* { *key: value* }	add `cookie` to user's cookies if this cookie doesn't exists and replace it otherwise

Finally, Technical Information Disclosure (TID) vulnerabilities refer to any technical information about the server. This can take many forms such as service banners indicating what technology is runs the server or stack traces when the server encounter errors.

4 EXPERIMENTS

4.1 SCWAD Framework

The SCWAD framework is designed as a Python package and use the playwright library (Microsoft, 2024) to instrument a web browser. Action selection for the automated agent follows a predefined strategy, implemented as functions returning elements from a list of feasible actions. For pentesting, the knowledge base can be initialized with only a starting URL, a set of users (that can be empty) and the session cookie name or a pre-filled database can be supplied. SCWAD also includes a replay mode, where it reads a filled database and replays each action sequentially. In this mode, SCWAD creates a new knowledge base in parallel to the input knowledge base. This allows for ensuring the new knowledge base contains the same information as the original knowledge base by the end of the replay, and comparing vulnerabilities found during the replay to the initial knowledge base to detect effective patches. Furthermore, the replay mode checks that actions it must do can be built from the new knowledge base as the impossibility to do so would indicate significant changes in the web application.

4.2 Pentesting UVVU with SCWAD

We develop the UVVU (Talon et al., 2023) web application to be used as a testbed to evaluate the effectiveness of the proposed SCWAD framework[1]. This application is poorly designed intentionally in order to embed vulnerabilities and mimic the real-world highly vulnerable applications.

[1]SCWAD will be made available after the publication

Experimental Setup: In these experiments, we set the three types of vulnerabilities in UVVU as described in Section 3.4: 7 technical information disclosure, 2 broken access control and 1 XSS. We test the 4 different strategies (Section 3.2) in the choice of actions to perform. The first three strategies serve as the baselines of the search strategies compared to *vulnerability tracking*. All of the 4 search strategies have the same pool of possible actions to make their choice for each step of the pentest process. For these strategies, an action is referred as a possible action to execute, when the arguments of the command can be filled with the records in the Knowledge Base or directly by the Oracle in the proposed system (see Figure 1). The strategies select one action from all the possible actions to execute based on their own criteria.

The first three search strategies directly select actions from the pool of all possible actions. In contrast, the vulnerability tracking strategy uses sequences of actions. These sequences are manually coded by human pentesters as a generalization, upon their experiences of pentest, of sequences that may trigger the different types of vulnerabilities (BAC, XSS, TID). Moreover, these sequences were defined without knowledge of the target web applications they would be tested on. These sequences are referred as action templates in the followings. For each step t of the pentest process, the vulnerability tracking module first compares the actions executed at the previous two steps ($t-1$ and t) with the corresponding steps in the action templates. If it can find an exact match in the action templates, this module will use the immediately next action at $t+1$ suggested by the exactly matched action template. Otherwise, it will adopt the *exploit-first selection* strategy. We compare the first three strategies to the vulnerability tracking module to demonstrate that fuzzers (random strategy) and automated scanners (explore-first and exploit-first strategies) cannot achieve the performances of pentesters to exploit the vulnerabilities that require to execute several actions in a chain.

We propose 5 metrics to measure the performances of the four different search strategies applied

Table 2: SCWAD performances on UVVU application.

Strategy	♯ update	♯ successful actions	♯ failed actions	Duration	Success rate
Random selection	55	54	6	4 min	50%
Explore first	69	68	4	5 min	70%
Exploit first	63	64	2	5 min	70%
Vulnerability tracking	73	74	17	6 min	100%

in our framework. *First*, we calculate the number of updates to the knowledge base resulting from the actions (referred as ♯ *update*). It indicates how many times new information were added to the knowledge base after the execution of an action completed with a success flag. More frequent updates correspond to richer information about the target application, thus indicating a better coverage over the potential vulnerabilities in pentested application. With this setting, a better search strategy is expected to induce more frequent update, allowing security analysts gain a better knowledge about the target application. *Second*, we computed the number (♯) of successful and failed actions. An action is considered successful when the web application executes it, otherwise it is considered failed. We emphasize that either an action is executed successfully or with a failure can equally bring the pentester more knowledge about the target application. For example, the success or failure of data injection to an URL and/or form indicates that XSS payload injection is feasible or infeasible to trigger further vulnerabilities by injecting payloads into the target server. This will guide the choice of the following actions to execute. One search strategy is better than another, if it can perform more actions (successful or failed) within the time limit. Either way can improve the pentester's knowledge about the target web application *Third*, we also gathered the execution time needed by SCWAD.*Finally*, we check and record as a success rate how many of the 10 vulnerabilities that each strategy manages to unveil. Table 2 presents the results of the four different strategies used in SCWAD to pentest UVVU.

The random selection strategy randomly selects an action from the list of possible actions, resulting in the poorest performance with fewer states reached by the knowledge base. Due to its inefficient exploration, it can only identify fifty percent of the vulnerabilities intended to be uncovered in the web application. The vulnerabilities it failed to uncover were related to technical information disclosure, which it couldn't reach due to inadequate state coverage.

The explore-first strategy navigates through all permitted pages before attempting any exploitation, while the exploit-first strategy immediately seeks to exploit vulnerabilities upon encountering a potentially exploitable state. This entails injecting forms and accessing restricted pages whenever they are incorporated into the knowledge base. Both strategies exhibited comparable performance, uncovering seventy percent of the vulnerabilities. Their broader coverage compared to the random selection strategy allowed them to gather more insights into the web application's configuration, reaching a greater number of states in the knowledge base. However, they also missed instances of technical information disclosure.

The suboptimal coverage of these three strategies underscores the significance of sequencing actions effectively in an uncontrolled environment, where each action may precipitate irreversible consequences for subsequent actions.

The vulnerability tracking strategy enhances its repertoire of potential actions by not only scrutinizing previously executed actions but also prioritizing them based on the exploit-first selection strategy. This approach constructs a logical sequence of actions akin to those performed by human pentesters, leading to vulnerability discovery. For instance, instead of directly attempting injections into a form or HTTP parameter, it initially searches for reflected values within these elements. Moreover, it endeavors to bypass identified server errors when exploiting vulnerabilities by experimenting with alternative payloads on the same injection point. By reaching states inaccessible to other strategies, this approach can uncover all vulnerabilities, leveraging more sophisticated exploitation techniques that provoke server errors in the UVVU application, thereby disclosing technical information such as stack traces. Figure 2 illustrates how this strategy streamlines the array of potential actions compared to others, resulting in superior outcomes. We emphasize that the correlation between the actions taken in the pentest process is crucial in efficiently discovering vulnerabilities. In a pentest process, strategically selecting actions that yield comprehensive information about the target application in earlier steps can swiftly limit the available actions in subsequent steps, leading to efficient vulnerability discovery with minimal actions. Consequently, an effective pentest search method is anticipated to involve a minimal number of actions throughout the process. Simultaneously, this method should rapidly decrease the number of potential actions for later stages of the pentest. We can note that vulnerability tracking strategy has the highest number of failed actions, this is because this strategy leverage such failures to cut short action sequences, participating in rapidly decreasing the number of possible actions. Let's take two examples to demonstrate how vulnerability tracking shrinks down the number of potentially available actions. First, in comparison to the explore-first strategy, trying to exploit an XSS may trigger server errors that would redirect the client to an error page. This means that failed exploitations may result in successful exploration, avoiding the need to specifically explore for these error pages. Second example compares to the exploit-first strategy. Reflected XSS vulnerabilities can only occurs if data entered by a user are reflected (directly or after some transformation) by the server. Then trying XSS injections on endpoints that does not reflect any data is useless. While the exploit-first strategy act like a fuzzer and tries every possible payload, the vulnerability tracking strategy first check if any data is reflected. If no data is reflected, the vulnerability tracking strategy cut short the search of XSS on that endpoint, effectively removing several possible actions that would try to inject payloads.

4.3 Comparison with State-of-the-Art Tools

Experimental Setup: To evaluate the performances of SCWAD compared to the state-of-the-art tools, we execute these tools against three different web applications. The first web application is Damn Vulnerable Web Application (DVWA, 2023) designed to assist security professionals in testing their skills and tools. This web application encompasses most major web application vulnerability types, offering three difficulty levels. We conduct tests at the "medium" difficulty level for all the scenarios. However, DVWA does not mirror a real-life web application accurately. Vulnerabilities are entirely isolated from one another and lack integration into authentic application features. The second web application is WackoPicko, introduced in (Doupé et al., 2010). It simulates a genuine commercial web application, akin to UVVU. Although it mimics a real application, it employs different technologies and was last updated in 2017. The last web application we examined is UVVU, detailed in Section 4.2. All three web applications implement at least one vulnerability of each type: technical information disclosure (now referred as TID), broken access control (BAC) and reflected XSS.

DVWA has 1 BAC, 18 reflected XSS and 17 TID. WackoPicko has 1 BAC, 2 reflected XSS and 2 TID. UVVU has 2 BAC, 1 reflected XSS and 7 TID. These numbers where given by the authors of each web application and we manually check for they are exploitable before running our experiment. They serve as the ground truth of what vulnerabilities the different tested tools should found when scanning or pentesting the applications.

We compare SCWAD to 3 state-of-the-art scanners that will serve as baselines for our experiments. The first tool is the scanner of BurpSuite Pro, a proprietary software developed by PortSwigger that is largely used by pentesters. The second is the scanner of ZAP, an open sourced tool from the OWASP also widely used by pentester. The last tool is Wapiti, an open source web scanner (Surribas, 2023). All these scanners were launched with their default configuration and best effort were made to allow them to perform authenticated scans.

To assess the performance of both SCWAD and the baseline scanners, we calculated three metrics. Firstly, the True Positive Rate (TPR) represents the ratio of correctly identified vulnerabilities to the total number of expected alerts. The number of expected alerts is determined by the pre-defined ground truth of vulnerabilities for each web application, while true positive alerts indicate instances where the tool correctly identifies a vulnerability corresponding to one of the ground truth vulnerabilities. Secondly, the False Positive Rate (FPR) measures the ratio of incorrectly identified vulnerabilities to the total number of expected alerts. False positive alerts are those that do not correspond to any ground truth vulnerability. Lastly, the False Negative Rate (FNR) represents the ratio of missed vulnerabilities to the total number of expected alerts. False negatives occur when the tool fails to detect a ground truth vulnerability, resulting in no alert being raised.

Each of the applied tool aims to detect the embedded vulnerabilities as much as possible, within the limited time of two hours. At the end of the experiments, all the 4 tools were able to finish before the time limit. However it is worth noting that the Wapiti fuzzer failed to scan DVWA due to its mishandling of anti-CSRF tokens during the login phase. The results of each scan are presented in Table 3, 4 and 5.

The results indicate that SCWAD is the sole tool to detect Broken Access Control vulnerabilities and it does it with 100% TPR. SCWAD is designed to manage multiple users and compare their interactions, enabling it to identify such vulnerabilities. In contrast, other tools are unable to unveil this type of vulnerability as they only handle sessions to broaden their web application crawling.

Regarding the reflected XSS vulnerabilities,

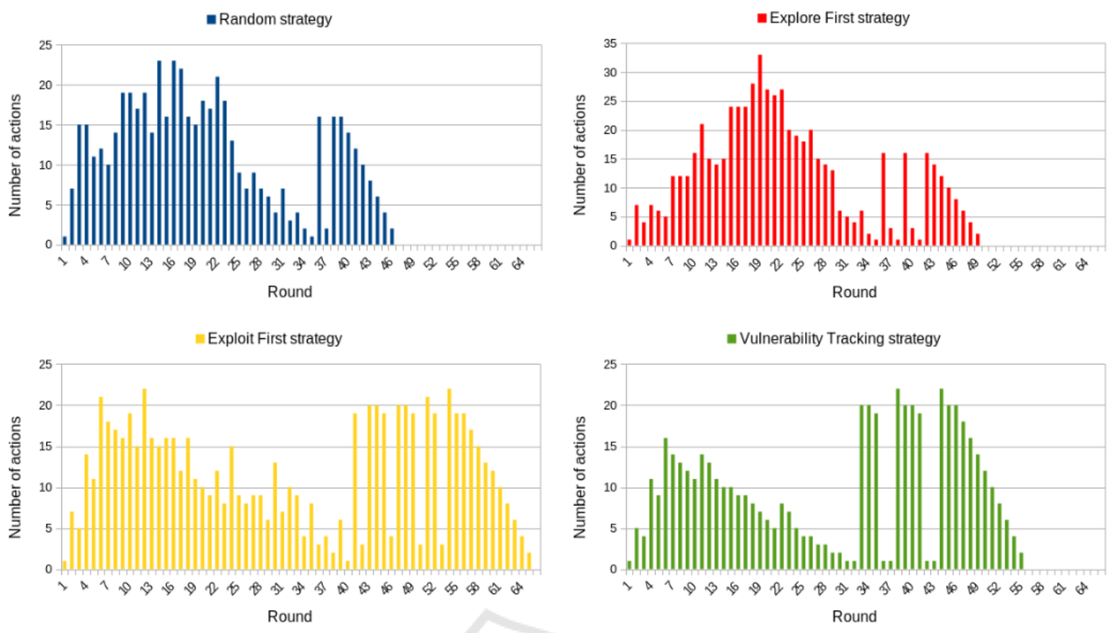

Figure 2: Number of possible actions built by SCWAD in each round for different strategies during a pentest over UVVU.

Table 3: XSS Detection: comparison between state-of-the-art tools
✗ means that the tool has crashed on the target app.

XSS detection			
Target App	DVWA	Wackopicko	UVVU
Existing vulns	18	2	1

Pentesting with BURPSUITE

TPR	100%	100%	100%
FPR	10%	33%	0%
FNR	0%	0%	0%

Pentesting with WAPITI

TPR	✗	50%	100%
FPR	✗	0%	0%
FNR	✗	50%	0%

Pentesting with ZAP

TPR	61%	50%	0%
FPR	0%	0%	0%
FNR	39%	50%	100%

Pentesting with SCWAD

TPR	72%	100%	100%
FPR	0%	33%	0%
FNR	28%	0%	0%

SCWAD performs similarly to BurpSuite Pro (the tool with highest TPR and lowest FNR on every application). On DVWA, SCWAD has a highest FNR, this discrepancy arises because SCWAD can not access certain parts of the application, which is accessible only through JavaScript events triggered by button clicks. This is a feature not yet implemented in SCWAD. But, BurpSuite has a highest FPR while SCWAD has 0% meaning BurpSuite generated false positives as it detects a reflected XSS because it can inject a script tag. But due to Content Security Policy, the injected script is not executed, which demonstrate

Table 4: Broken Access Control Detection: comparison between state-of-the-art tools.

Detection of BAC			
Target App	DVWA	Wackopicko	UVVU
Existing vulns	1	1	2

Pentesting with BURPSUITE
no broken access control discovered

Pentesting with WAPITI
crashed on DVWA
no broken access control discovered on others

Pentesting with ZAP
no broken access control discovered

Pentesting with SCWAD

TPR	100%	100%	100%
FPR	0%	0%	0%
FNR	0%	0%	0%

Table 5: Technical Information Disclosure: comparison between state-of-the-art tools

Detection of TID			
Target App	DVWA	Wackopicko	UVVU
Existing vulns	17	2	7
Pentesting with BURPSUITE			
no technical information disclosure discovered			
Pentesting with WAPITI			
crashed on DVWA			
no technical information disclosure discovered			
Pentesting with ZAP			
TPR	100%	100%	14%
FPR	0%	80%	0%
FNR	0%	0%	86%
Pentesting with SCWAD			
TPR	18%	100%	100%
FPR	0%	0%	0%
FNR	83%	0%	0%

the effectiveness of this protection mechanism. In the case of WackoPicko, ZAP and Wapiti fail to identify all the reflected XSS, while SCWAD and BurpSuite detect an extra one (FPR of 33%). The additional reflected XSS identified by SCWAD and BurpSuite is a misclassified stored XSS.

Regarding technical information disclosure vulnerabilities, SCWAD outperforms the other tools overall. BurpSuite and Wapiti can not generate reports for such vulnerabilities. ZAP yields inconsistent results with a TPR of 100% on DVWA but only 14% on UVVU and a FPR of 80% on WackoPicko (ZAP mistakes stack traces for absolute paths on the server due to a directory traversal vulnerability). Even though it has a low TPR on DVWA (due inaccessible parts of the web application and TID in errors that were not triggered), SCWAD triggers errors with stack traces on UVVU that ZAP does not identify.

Our experiment demonstrates that SCWAD can identify BAC vulnerabilities that the other tools miss. Additionally, SCWAD produces fewer reports than traditional scanners, focusing solely on genuine vulnerabilities and avoiding false positives.

We also tested these different tools on the ginandjuice.shop (gin,) to see how they behave on an uncontrolled web application. Ginandjuice.shop is a web application design by PortSwigger (Burpsuite's editors) to test scanners. Because scanners are not able to detect broken access control vulnerabilities, this web site does not implement any. It does not implement technical information disclosure as well, so only XSS vulnerabilities remain to be found. Obviously Burpsuite find all XSS vulnerabilities but Wapiti found none (0% TPR) and ZAP only one (25% TPR) and generate a false positive (25% FPR). On the other hand SCWAD find two of the four XSS vulnerabilities (50% TPR). The missed ones are because the four payloads generated by the Oracle do not trigger the vulnerability, hence enriching the Oracle would be enough to find these vulnerabilities.

4.4 Pentesting Real World Applications with SCWAD

Before testing SCWAD on real world applications, we tried it on truly black boxed web applications. To do so we launched it against challenges web applications of the (RootMe, 2023) platform. SCWAD discovered stored XSS on 4 different challenges.

To scan real world web applications without disturbing them, we prevented SCWAD from performing possibly harmful actions. This means, we removed any call to the Oracle and so any payload injection. This way, we were only looking for Broken Access Control and Technical Information Disclosure vulnerabilities. In addition to removing payload injections, we limited the rate of SCWAD (at most 1 action every 0.8 seconds) to avoid overloading the server and prevent unwanted denial of service.

We selected 3 web applications on which we were able to create two accounts. We then have three users on each web application, two different authenticated users and the unauthenticated one. This is necessary to compare what these users can access and find BAC vulnerabilities if any.

In one hour we are able to execute between 250 and 300 actions depending on the loading time of the web site.

Within more time, we can execute more actions but we were limited by servers protections that limited our rate (766 actions max within 7 hours). We are able to get very huge amount of pages (2.1K, 3.9K) which make our processing time very high, but our way of defining pages is necessary to find BAC. To reduce our processing time, we believe we need to do page clustering and "understand" the data.

We find a technical information disclosure on one web application (service banner). This result is not surprising as such real world web applications are usually pentested and patched before being released in production. Technical information disclosure vulnerabilities are not critical vulnerabilities and are often not considered harmful by application owners that may not put effort in patching them.

Ethical Considerations. During the realization of this work, we don't upload any payloads to the target real-world web applications. We only submit queries and observe the returned value. Our aim is to evaluate whether SCWAD can be used in practices and avoid poisoning the real-world web applications. We unveil a TID vulnerability in the Crazygames application. We have shared this unveiling with the application owner, explaining our experiments are designed for scientific research only.

5 CONCLUSION

In this study, we have introduced SCWAD, an automated web application pentesting framework. Central to our approach is the structured and quantifiable representation of a target web application's elements, referred to as the knowledge base in this context. The introduction of the knowledge base concept brings forth significant advantages. Firstly, it enables the pentest process to be modeled as a sequential decision-making problem. SCWAD incorporates an automated pentest agent that selects viable vulnerability exploration actions based on the attributes defined in the knowledge base. Additionally, the pentest agent within SCWAD can enhance the understanding of the target web application by updating attribute values in the knowledge base, thereby guiding subsequent vulnerability exploration actions. Secondly, the design of the knowledge base allows vulnerabilities to be encoded as logic expressions involving the attributes of the knowledge base. This feature facilitates interaction between the automated pentest agent and human oracles; the agent can assess potential vulnerabilities' feasibility by matching knowledge base attributes with encoded logic expressions representing vulnerability signatures. Looking ahead, our future research aims to integrate reinforcement learning-based agents, enhancing adaptability and efficiency in vulnerability exploration. The pentest policies learned through interactions with diverse web applications can empower human security analysts to uncover novel vulnerability exploitation methods. This knowledge, in turn, can inform proactive measures for strengthening the security posture of target web applications.

REFERENCES

Ginandjuice.shop. https://ginandjuice.shop.

Buyukkayhan, A. S., Gemicioglu, C., Lauinger, T., Oprea, A., Robertson, W., and Kirda, E. (2020). What's in an exploit? an empirical analysis of reflected server XSS exploitation techniques. In *RAID 2020*.

Dahse, J. and Holz, T. (2014). Simulation of built-in php features for precise static code analysis. In *NDSS 2014*.

Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security*.

Doupé, A., Cova, M., and Vigna, G. (2010). Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131.

Drakonakis, K., Ioannidis, S., and Polakis, J. (2023). Rescan: A middleware framework for realistic and robust black-box web application scanning. In *NDSS*.

DVWA (2023). Damn vulnerable web application. https://github.com/digininja/DVWA.

Eriksson, B., Pellegrino, G., and Sabelfeld, A. (2021). Black widow: Blackbox data-driven web scanning. In *IEEE S&P 2021*.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: a static analysis tool for detecting web application vulnerabilities. In *IEEE S&P 2006*.

Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. Secubat: A web vulnerability scanner. In *World Wide Web Conference*.

Microsoft (2024). Playwright.

Nashaat, M., Ali, K., and Miller, J. (2017). Detecting security vulnerabilities in object-oriented php programs. In *SCAM 2017*.

Nunes, P., Medeiros, I., Fonseca, J. C., Neves, N., Correia, M., and Vieira, M. (2018). Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*.

Nunes, P. J. C., Fonseca, J., and Vieira, M. (2015). phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In *DSN 2015*.

OWASP (2021). Open source foundation for application security top 10 - 2021. https://owasp.org/Top10/.

OWASP (2023a). Owasp. https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html.

OWASP (2023b). Zed attack proxy (zap). https://www.zaproxy.org/.

Pellegrino, G., Tschürtz, C., Bodden, E., and Rossow, C. (2015). JÄk: Using dynamic analysis to crawl and test modern web applications. In *RAID 2015*.

PortSwigger (2023). Burp suite's web vulnerability scanner. https://portswigger.net/burp/vulnerability-scanner.

RootMe (2023). Root me. https://www.root-me.org.

Surribas, N. (2023). Wapiti. https://wapiti-scanner.github.io/.

Talon, N., Viet Triem Tong, V., Guette, G., and Han, Y. (2023). Uvvu. https://github.com/scwaduvvu/uvvu.

Zhang, B., Li, J., Ren, J., and Huang, G. (2022). Efficiency and Effectiveness of Web Application Vulnerability Detection Approaches. *ACM Computing Surveys*.