SLVHound: Static Detection of Session Lingering Vulnerabilities in Modern Java Web Applications

Haining Meng
SKLP
Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
menghaining@ict.ac.cn

Jie Lu*
SKLP
Institute of Computing Technology, CAS
Beijing, China
lujie@ict.ac.cn

Yongheng Huang
SKLP
Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
huangyongheng20s@ict.ac.cn

Lian Li*
SKLP
Institute of Computing Technology, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
lianli@ict.ac.cn

Abstract

Session Lingering Vulnerability (SLV) is an often overlooked authentication flaw that allows sessions to persist after authentication-sensitive operations. Despite its widespread occurrence and severe impact, SLVs have received little attention. To address this gap, we present the first comprehensive study of SLV in Web applications, introducing a novel detection tool called SLVHound. Our approach employs static analysis of both code and SQL queries to identify authentication-sensitive operations and session expiration. SLVHound then detects SLVs by verifying whether authentication-sensitive operations are consistently followed by session expiration.

We evaluated SLVHound on 15 popular Web applications, uncovering 46 potential vulnerabilities. Further analysis confirmed 44 of them as true SLVs, including 30 previously unreported vulnerabilities, with 16 CVE IDs granted.

CCS Concepts

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

ACM Reference Format:
Haining Meng, Jie Lu, Yongheng Huang, and Lian Li. 2025. SLVHound: Static Detection of Session Lingering Vulnerabilities in Modern Java Web Applications. In *the 16th International Conference on Internetware (Internetware 2025), June 20–22, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3755881.3755895

1 Introduction

Web applications, which handle sensitive data such as medical records and personal identity information, are prime targets for cyber-attacks. To protect this data, web applications employ authentication mechanisms [6] to ensure that only verified users can access the system. These mechanisms authenticate users by verifying credentials like usernames, passwords, and digital certificates.

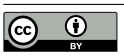
To enhance user experience and system performance, web applications implement session mechanisms to avoid frequent credential verification. After initial authentication, these mechanisms generate a unique session identifier stored on the server side. This session identifier is then included in subsequent requests, enabling continuous user identification without requiring repeated authentication.

However, session mechanisms introduce several security risks, as illustrated by the following attack scenarios:

- **Session Hijacking Scenario**: An attacker obtains a user's active session identifier through Cross-Site Scripting (XSS) [45] and Cross-Site Request Forgery (CSRF) [44] attacks, without knowing the actual credentials. When the victim notices suspicious activities and changes their password as a precautionary measure, the hijacked session must be expired immediately; otherwise, the attacker maintains unauthorized access despite the password change.
- **Credential Compromise Scenario**: An attacker gains access by obtaining the victim's actual credentials (e.g., through password theft or brute force). When the victim or administrator responds by changing the password or disabling or deleting the account to stop the unauthorized access, the attacker's existing sessions must be expired; otherwise, these critical security measures become completely ineffective.

Given these security risks, web applications must be particularly vigilant when users perform *authentication sensitive operations (ASOs)*, such as password modifications or account disabling. These operations fundamentally alter the authentication state of an account and, considering the possibility of compromised sessions, necessitate the immediate expiration of all active sessions to prevent attackers from continuing their malicious activities.

*Corresponding author



Internetware 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1926-4/25/06
https://doi.org/10.1145/3755881.3755895

However, web applications do not always properly expire sessions following ASOs, resulting in vulnerabilities we term *Session Lingering Vulnerability (SLV)*. Our analysis (§2) of authentication-related vulnerabilities from 2020 to 2023 demonstrates both the prevalence and significant security impact of SLVs. The combination of easily exploitable session hijacking vectors (XSS, CSRF) and SLVs creates a particularly dangerous security gap. Consequently, detecting SLVs in Web applications has become critically important for maintaining robust security postures.

Manual dynamic testing methods, which involve executing applications with test inputs to observe runtime behavior, currently serve as the primary approach for identifying SLVs. However, such methods demonstrate significant limitations in detecting SLVs within modern web applications. Manual dynamic testing fundamentally relies on predefined test cases crafted by human testers, whose effectiveness is constrained by both the tester's expertise and the inherent difficulty in achieving complete execution path coverage, particularly in applications implementing sophisticated authentication mechanisms. For instance, RuoYi [62] implements password modification through two distinct APIs (for administrators and users), while CSKefu [12] provides separate APIs for desktop and mobile platforms, each performing ASOs requiring session expiration. To effectively test for SLVs, manual dynamic testing would need to identify all APIs containing authentication-sensitive operations, execute these APIs, and subsequently verify proper session expiration—a process that becomes increasingly impractical as application complexity grows. In contrast, static analysis techniques can effectively address this limitation by systematically examining multiple execution paths to verify the presence of session expiration on paths containing ASOs, making static analysis crucial for detecting SLVs effectively.

However, detecting SLVs through static analysis presents significant challenges. Unlike traditional taint-type vulnerabilities (e.g., XSS or SQL injection) which exhibit clear data flows, SLVs are application logic vulnerabilities that require semantic understanding. This semantic nature of SLVs manifests in two critical aspects: authentication data processing and session management implementation. Specifically, detecting SLVs requires identifying authentication data, classifying its modifications as ASOs, and understanding how session management expires sessions.

The semantic complexity of SLV detection is further amplified by two factors. First, the implementation of authentication and session management varies significantly between different applications, making it challenging to develop universal detection patterns. Second, the effectiveness of static analysis heavily depends on the specific coding practices employed in each application. Consequently, the detection of SLVs through static analysis remains largely unexplored and presents significant technical challenges.

To address the aforementioned challenges, we conducted an empirical study on 72 SLVs, gaining insights that guided us in resolving these challenges. We found that despite the varied implementations of authentication across different applications, the authentication data is always stored in databases. For example, passwords used for authentication are typically stored in a column of the `User` table. Therefore, modifications to the `password` column of `User` tables are ASOs that require session expiration. Our study also identified two primary approaches to session management in Web applications. The first approach involves developers creating custom session management solutions, typically by storing sessions in a dedicated *session* table within a database. In this scenario, session expiration is handled by deleting the corresponding entries from this table. The second approach relies on authentication frameworks, such as Spring Security [47], which offer built-in APIs like `SessionInformation.expireNow()` to expire sessions effectively.

Based on our findings, we developed SLVHound, a static analysis tool to identify SLVs in Java Web applications. SLVHound operates through three structured steps. The first step employs static analysis techniques and SQL queries to identify database columns storing authentication data, as well as to identify all modifications to these columns, which are considered ASOs. In the second step, SLVHound, guided by the authentication framework, identifies specific API calls responsible for session expiration. This phase also applies heuristic rules to determine if the application manages sessions itself and locates the *session* table, treating DELETE operations on this table as session expiration. The third step involves inter-procedural data flow analysis to detect SLVs by identifying execution paths that include an ASO without a session expiration.

Our experiments on 15 Java open-source Web applications demonstrate that SLVHound can accurately identify ASOs and session expiration within these applications. It successfully detected and reported 46 SLVs, including 30 previously unreported vulnerabilities, resulting in 16 new CVE IDs, with only 2 false positives.

In summary, our contributions are as follows:

- **Empirical Study**: We conducted the first empirical study on *Session Lingering Vulnerabilities (SLVs)* in Web applications, pioneering research in this area.
- **SLVHound**: Drawing from findings gained during our empirical study, we designed and implemented SLVHound, the first tool specifically for detecting SLVs. We applied this tool to 15 applications and identified 44 true SLVs.
- **Open Source**: To support research and future security improvements, we have open-sourced SLVHound's codebase, including the tool and all studied and detected vulnerabilities. All resources are available at SLVHound.

2 Empirical Study

This section presents our approach for collecting and analyzing authentication-related vulnerabilities, with a specific focus on SLVs. Furthermore, through source code analysis, we examine the implementation of authentication and session management mechanisms in Web applications containing SLVs.

2.1 Collecting and classifying vulnerabilities

To gain a more comprehensive understanding of SLVs, we conducted an extensive analysis of the authentication-related vulnerabilities in open-source projects. Our study focused on vulnerabilities reported between January 2020 and December 2023, as documented in the National Vulnerability Database (NVD). We employed the keywords and Common Weakness Enumeration (CWE) types outlined in Table 1 as our search criteria. This initial query yielded 2,900 vulnerabilities from NVD. Subsequently, we conducted a manual review and filtering process, which resulted in the identification of 330 relevant authentication vulnerabilities. In addition, to deepen

Table 1: Search criteria in the vulnerability collection process

Keywords	authentication, session
CWEs	CWE-287, CWE-295, CWE-297, CWE-303, CWE-304, CWE-306, CWE-307, CWE-1390, CWE-613, CWE-288, CWE-425, CWE-322, CWE-620, CWE-640, CWE-384

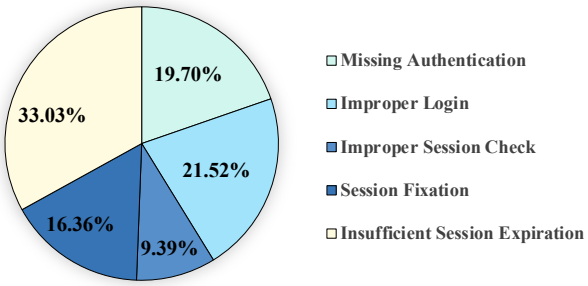


Figure 1: The distribution of authentication vulnerability

our understanding of the identified SLVs, we also manually analyzed the source code of these applications contained SLVs, with a particular focus on authentication and session management mechanisms.

We categorize the identified vulnerabilities into five distinct types, as illustrated in Figure 1. The first type, *Missing Authentication*, refers to the absence of necessary authentication for critical functions. The second category, *Improper Login*, encompasses vulnerabilities such as unlimited login attempts and the ability to infer password length based on login execution time. The third type, *Improper Session Check*, involves flawed session validation, including flaws in session verification logic that may lead to the acceptance of invalid sessions. *Session Fixation*, the fourth category, occurs when old sessions persist even after a user logs in again. Finally, *Insufficient Session Expiration* refers to the failure to properly expire sessions, encompassing SLVs.

2.2 Threats to Validity

Internal Validity:

Our vulnerability collection process relies on publicly reported issues from NVD, which may not capture vulnerabilities that were fixed without public disclosure or those reported through other channels. Additionally, while we employed systematic search criteria using carefully selected keywords and CWE types, some relevant vulnerabilities might have been missed if they were labeled differently or described using alternative terminology.

External Validity: We analyzed 330 authentication-related vulnerabilities collected between January 2020 and December 2023. While this represents a substantial dataset, it may not capture all possible variations of authentication vulnerabilities. However, we believe our dataset is representative because: (1) it spans diverse application types including blogs, content management systems, and enterprise applications; (2) the SLVs come from 59 different applications rather than being concentrated in a few projects; and (3) our systematic collection methodology using NVD provides broad coverage of publicly reported issues.

Construct Validity: Our manual review and categorization of vulnerabilities into five distinct types might be subject to researcher bias. To mitigate this, we conducted thorough code analysis of the vulnerable projects and cross-validated our categorization through multiple rounds of review. The classification was based on well-established security concepts and CWEs.

2.3 Findings

Finding 1: 21.82% of authentication vulnerabilities are due to SLVs, which actually represents a significant portion in real-world.

As shown in Figure 1, our study identified 330 authentication vulnerabilities, among which insufficient session expiration [8] constitute 33.03% of all cases. These can be categorized into two subtypes:SLVs and timeout configuration failures. Timeout configuration failures refer to cases where session expiration periods are set inappropriately long.

Notably, SLVs account for 66.06% of session expiration vulnerabilities (21.82% of total authentication vulnerabilities), while the remaining 33.94% stem from excessive timeout settings. This distribution reveals that despite extensive prior research on timeout-related vulnerabilities [14, 22, 54, 55], SLVs remain significantly understudied despite their higher prevalence (66.06% vs. 33.94%) and critical security implications.

Finding 2: Most (69.01%) SLVs have a high-security impact according to CVSS v3 scores, with 29.6% reaching a critical impact.

Of the 72 identified SLVs, 71 have assigned CVSS v3 scores, with 29.58% (21 vulnerabilities) rated as critical and 39.43% (28 vulnerabilities) as high severity. The mean CVSS v3 score of 7.7, which is notably higher than the typical severity scores in general vulnerability databases (5.0-6.0), further emphasizes the substantial security risk posed by these vulnerabilities. This severity distribution indicates that SLVs frequently result in significant security breaches when exploited, warranting immediate attention from the security community.

Finding 3: Authentication comprises multiple checkers distributed across a filter chain. These checkers verify the authentication data stored in databases to confirm whether a user can be successfully authenticated.

Based on our study of 59 Web applications containing SLVs, we observed that authentication in modern Java Web applications is frequently implemented using the chain of responsibility pattern [60], consisting of a series of specialized filters and login methods. Web frameworks, such as Spring, provide built-in implementations of this pattern, offering developers a convenient way to utilize it. Take the filter chain in Figure 2 for example, it includes filters and a login method. $Filter_1$ checks whether the user is an administrator to confirm if they have permission to access a specific Web URL. If yes, $Filter_2$ is involved to check if the user is forbidden from logging in.

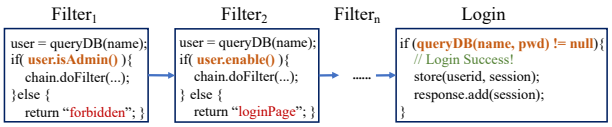


Figure 2: Authentication checkers distributed in the filter chain.

If all filters pass, the login method in Figure 2 is invoked, using a checker to validate the user-supplied `username` and `password`.

Additionally, as shown in Figure 2, the filter and login utilize data from the incoming request, such as the `username`, to query authentication data (i.e., the `password`) from the corresponding authentication column. They then determine whether the user can log in by verifying the queried authentication data.

Finding 4: 43.86% SLVs are due to lacking session expiration in custom session implementations, which typically store session data in databases, while 56.14% are due to lacking session expiration in framework-based session management.

Web applications employ two primary methods for managing authentication sessions: custom implementations and framework-based approaches. In custom implementations, developers, based on specific application requirements and security needs, create their own session management logic, with sessions commonly stored in databases or memory caches. For example, as illustrated in the login method of Figure 2, after verifying the `username` and `password`, the `session` and its associated `userID` are stored in a database table. This approach enables session expiration through direct deletion of session entries from the storage system. Alternatively, many Web applications utilize framework APIs for unified session management. For instance, Spring Security's `SessionRepository` interface provides standardized methods such as `save` and `deleteById` for storing and expiring sessions, respectively.

3 SLVHound

Based on Finding 1 and Finding 2, SLVs are both widespread and severely impactful, underscoring the critical need for their detection. To address this, we developed a static analysis tool, SLVHound, which operates through a three-step process:

(1) **ASOs identification:** Leveraging Finding 3, we employ comprehensive static analysis to precisely pinpoint all authentication checkers within a Web application. These checkers facilitate the systematic identification of authentication columns that store authentication data and the authentication-sensitive operations (ASOs) that modify these columns.
(2) **Session expiration identification:** Informed by Finding 4, this step systematically focuses on identifying and analyzing session expiration mechanisms from both standard authentication frameworks and custom session management implementations in real-world Web applications.
(3) **SLV Detection:** The final step of SLVHound employs inter-procedural data flow analysis to verify the existence of execution paths where ASOs occur without corresponding session expiration, thereby detecting potential SLVs.

```
1 {"role": "system", "content": "You are an experienced
  ↪ and professional Java programmer with extensive
  ↪ experience in Web project development. Your task is
  ↪ to analyze Java code snippets and determine if they
  ↪ represent user login methods in a Java Web
  ↪ project."},
2 {"role": "user", "content": "Analyze the following Java
  ↪ method and determine if it is a user login method in
  ↪ a Java Web project. Your response must be EXACTLY
  ↪ 'yes' or 'no', without any additional explanation or
  ↪ commentary. Here is the method code:[]"}
```

Figure 3: Prompt used to identify login methods.

```
1 javax.servlet.Filter.doFilter()
2 org.spring...HandlerInterceptor.preHandle()
3 org.spring...HandlerInterceptorAdapter.preHandle()
4 org.spring...HandshakeInterceptor.beforeHandshake()
5 org.spring...OncePerRequestFilter.doFilterInternal()
6 org.spring...BasicAuthenticationFilter.doFilterInternal()
```

Figure 4: Filter methods provided by Web framework .

3.1 ASOs Identification

Guided by Finding 3, we have developed an automated three-step process to identify ASOs that modify authentication data. This process consists of: (1) identifying authentication checkers within filters and login methods, (2) inferring the authentication columns utilized by these checkers, and (3) locating ASOs that change these authentication columns.

3.1.1 Identify authentication checkers. As shown in Figure 2, authentication checkers are typically found within two primary methods of Web applications: filters and login methods. Therefore, the initial step in identifying these checkers involves pinpointing these specific types of methods.

Identifying login. Identifying the login method presents a challenge, as Web applications often feature multiple login methods to support various functionalities and endpoints. For example, CSKefu [12] has a total of three login methods, one for API requests and two for Web page requests. Manual specification of these methods can be tedious and prone to oversight. To address this, we design an automated approach that combines static analysis with LLM-based verification. First, we traverse the call graph and examine each method node to identify potential entry points - methods that have no callers in the application. Since login methods are typically among these entry points, this step creates an initial candidate set. Subsequently, we combine these candidate methods (including their signatures and bodies) with our designed prompt in Figure 3, and leverage a Large Language Model (LLM) due to its proven capability in code understanding and pattern recognition [15, 31, 36, 46]. The prompt is carefully crafted to elicit binary (yes/no) responses, ensuring consistent and deterministic results, which allows us to effectively filter out actual login methods from these entry points.

```
javax.servlet.http.Cookie.<init>()
javax.servlet.http.HttpSession.setAttribute()
javax.servlet.http.HttpServletResponse.addHeader();
org.springframework.http.ResponseEntity.<init>()
org.springframework.http.ResponseEntity.ok()
org.springframework.http.HttpHeaders.add()
```

Figure 5: Response writing methods provided by Web framework .

Identify authentication filters. In modern Web application development, filter methods are essential components typically provided by Web frameworks, as shown in Figure 4. Developers often customize these methods by overriding them to implement specific authentication logic. This standardized approach allows for systematic identification of all filters within the application.

Identify authentication checkers. Subsequently, we analyze all check statements within filters and logins to determine their relevance to authentication, as not all check statements are authentication checkers. For instance, some checkers merely verify the client type from which a request originates, without impacting authentication outcomes. To accurately identify genuine authentication checkers, we employ a custom-designed heuristic rule. This rule is based on the observation that successful authentication typically triggers specific actions, such as executing the next filter by calling the filter method(i.e. `doFilter` in Figure 2) or appending something to the response by calling the response writing method(i.e. `response.add` in Figure 2).

Filter methods and response writing methods are illustrated in Figure 4 and Figure 5 respectively. After specifying these methods, we employ a control-dependency approach to identify authentication checkers. Specifically, we consider a checker to be authentication-related if the callsites of these specified methods are control-dependent on it. For example, as illustrated in Figure 2, the three `if` statements satisfy this condition, thereby qualifying as authentication checkers.

3.1.2 Inferring authentication columns. Our approach to identifying authentication columns involves a two-step analysis. First, we employ data flow analysis to determine the set Q_R of SQL queries that can be reached by request data. Subsequently, we conduct further data flow analysis to identify the set B_Q of if-branches that utilize the results of these queries, denoted as R_Q. The set of authentication columns is then defined as the union of two subsets: columns used in the *WHERE* clauses of queries in Q_R, and columns used in the check conditions in B_Q that data-depend on R_Q. This comprehensive definition encompasses both direct query-based authentication checks and conditional checks based on query results. The former category serves as an exemplar of the checker in Filter1, while the latter represents the checker in the login method, both of which are depicted in Figure 2.

ORM. A significant challenge in implementing the aforementioned method lies in identifying Q_R within the application code.

```
1  public void changePassword(String pwd, Stirng newpwd){
2     User user = res.findByUsernameAndPassword(name,
      ↪  pwd);//ORM operation
3     user.setPassword(newpwd);
4     res.save(user);//ORM operation
5  }
6  public interface UserRepository extends
   ↪  JpaRepository<User, String> {
7     User findByUsernameAndPassword(String username,
      ↪  String password);
8  }
9  @Entity
10 @Table(name = "cs_user")
11 public class User implements java.io.Serializable {
12     private String username;
13     private String password;
14 }
```

Figure 6: A code example of using the SpringDataJPA ORM framework.

Modern Web applications extensively utilize Object-Relational Mapping (ORM) frameworks, which generate SQL queries from ORM operations based on predefined SQL transformation rules.

Figure 6 illustrates a code example demonstrating how the ORM framework SpringDataJPA [59] transforms ORM operations into SQL queries according to its predefined rules. In this example, the *User* class is mapped to the *cs_user* table via the *@Table* annotation, with class fields corresponding to table columns. ORM operations on this table can be performed using methods like *findByUsernameAndPassword* in the *UserRepository* interface, which extends *JpaRepository*. According to SpringDataJPA transformation rules, the method name *findByUsernameAndPassword* indicates a SELECT operation (denoted by "find"), with the remaining part determining the columns involved ("username" and "password"). The "By" keyword denotes a WHERE clause. Based on these rules, SpringDataJPA translates the ORM operation in line 2 into an SQL query:

```
SELECT * FROM cs_user WHERE username = ? AND password = ?
```

This abstraction provided by ORM frameworks, while enhancing developer productivity, poses significant challenges for static analysts attempting to directly examine and analyze the underlying SQL queries.

ORM parser. To address the challenge of identifying SQL queries obscured by ORM frameworks, we have developed parsers for various ORM implementations. These parsers, leveraging the SQL transformation rules specific to each ORM framework, convert ORM operations into a standardized quadruple representation: $\langle OP, Table, Columns, Condition \rangle$, representing it as an SQL query. `OP` denotes the operation type (INSERT, DELETE, UPDATE, or SELECT), `Table` indicates the affected database table, `Columns` enumerates the modified columns, and `Condition` specifies the columns used either in the WHERE clause of the SQL query or in conditional statements within the application code.While this simplified representation omits certain details, such as the new values

used in UPDATE operations, it retains sufficient information for our primary objective: the identification of authentication columns and subsequent ASOs.

We demonstrate how our parser infers the four-tuple representation using the code example in Figure 6.

Inferring Operation Type Based on SpringDataJPA transformation rules, the `find` method in ORM operations corresponds to a SELECT operation (line 2), while `delete` represents DELETE operations, and `save` indicates either UPDATE or INSERT operations. To distinguish between UPDATE and INSERT when encountering a `save` operation, we employ def-use analysis by tracing the origin of the object being persisted. If the object is instantiated via a `new` instruction, the operation is classified as INSERT. Conversely, if the object originates from a database query, it signifies an UPDATE operation. In our example, since the `user` object at line 4 is retrieved from the database (line 2), we determine that the operation type is UPDATE.

Inferring Table As shown in line 10 of Figure 6, the table name is specified through the `@Table` annotation. Our parser identifies the corresponding table using this convention and associates the entity class, in this case `User`, as the table representative throughout the application code. This approach facilitates subsequent static analysis procedures.

Inferring Column To identify modified columns, we employ backward data dependency analysis. For instance, having established that line 4 represents an UPDATE operation, we analyze preceding modifications to the `user` object. At line 3, the `setPassword` method modifies the `password` column. The column identification follows SpringDataJPA naming conventions, where the column name follows the `set` prefix in setter methods.

Inferring Condition Following SpringDataJPA conventions, conditions are determined by the elements following the `By` keyword in ORM operation method names. Examining line 2, the method `findByUsernameAndPassword` indicates that `username` and `password` columns serve as condition columns in the query.

Discussion. Our custom-developed parsers effectively trace and analyze database operations in applications utilizing ORM frameworks and other persistence technologies. The current implementation supports six widely-adopted Java persistence solutions: traditional ORM frameworks (MyBatis [10], SpringDataJPA [59], Jakarta Persistence [18], and Hibernate [9]), as well as NoSQL database clients that provide similar object-to-database mapping functionality (MongoDB [40] and Redis [32]). Despite the technical differences between these technologies, our approach successfully bridges the abstraction gap that typically obscures direct query visibility. Each parser implementation is remarkably concise, averaging only 96 lines of code per framework, demonstrating the efficiency of our approach.

While the initial development of these parsers requires manual effort, we have designed a plugin-based architecture with well-defined interfaces that significantly simplifies the process of adding support for new ORM frameworks. Our system provides a standardized Parser interface that developers can implement to integrate additional ORM frameworks without modifying the core analysis. We have adopted an open-source development model that enables community participation and continuous improvement through this extensible architecture. This collaborative approach not only ensures the sustainability of the project but also facilitates the expansion of framework support through community contributions. Each new parser implementation only needs to map framework-specific ORM operations to our standardized quadruple representation, maintaining consistency in the analysis results across different frameworks. The complete parser implementations and extension documentation are available in our public repository for reference and community enhancement.

```
1  javax.servlet.http.HttpSession.invalidate()
2  org.apache.shiro.session.Session.stop()
3  org.apache.shiro.subject.Subject.logout()
4  org.apache.shiro.mgt.DefaultSecurityManager.logout()
5  org.apache.shiro.session.mgt.DefaultSessionManager.delete()
6  org.apache.shiro.session.mgt.eis.SessionDAO.delete()
7  org.spring...session.SessionRepository.deleteById()
8  org.spring...SpringSession...removeSessionInformation()
9  org.spring...SessionInformation.expireNow()
```

Figure 7: Session expiration APIs provided by Web frameworks.

3.1.3 Locating ASOs. After identifying the authentication columns, SLVHound locates all SQL queries that modify these columns, classifying them as ASOs. The modifications primarily consist of UPDATE and DELETE operations, while INSERT operations are excluded from this category. For UPDATE operations, SLVHound verifies that the modified columns are indeed authentication columns. In the case of DELETE operations, SLVHound confirms that the target tables contain authentication columns.

Furthermore, it is crucial to invalidate user sessions upon logout. To achieve this, we employed a method similar to the one illustrated in Figure 3 to identify all logout methods across the application codebase. We then treat the exit block of each logout method as an ASO, facilitating subsequent detection.

3.2 Session Expiration Identification

Session expiration in web applications encompasses both framework-based and custom implementations. For framework-based session expiration, our tool identifies specific API calls, with the relevant APIs illustrated in Figure 7. In custom implementations, we employ a two-fold approach: first, inferring session tables by analyzing post-login operations, and then identifying all DELETE operations on these tables, treating these operations as session expiration.

The comprehensive rules for inferring session tables are depicted in Figure 8. In these rules, the method `writeResponse`, as defined in Section 3.1.1, is specifically designated to add data to the response stream. `SQLSave` represents the ORM operation, derived in Section 3.1.2, responsible for saving data to the persistent database storage. It is crucial that `writeResponse` post-dominates `SQLSave`, meaning that whenever `SQLSave` is executed, `writeResponse` must also be executed in the control flow. The alias condition indicates that both methods operate on the same object instance, which is used to determine conclusively that these modifications are specifically applied to the session .

$$\frac{\begin{array}{c} writeResponse(a_1, ...), \\ SQLSave \leftarrow (OP, Tab, Columns, \{a2...\}), \\ OP \in \{UPDATE, INSERT\}, \\ writeResponse \textbf{ Post-dominate } SQLSave, \\ alias(a_1, a_2), \end{array}}{Tab \in S_{tables},}$$

Figure 8: Session tables inference rule.

3.3 SLV Detection

To detect SLVs in Web applications, we first construct a call graph. This graph serves as the foundation for creating an Interprocedural Control Flow Graph (ICFG). Utilizing the ICFG, we then apply Interprocedural Data Flow Analysis (IDFS) techniques to effectively identify SLVs.

Building Call Graph. Constructing a call graph typically requires specifying entry points, often using the *main* method. However, Web applications are framework-driven and lack a main method. Instead, entry points in Web applications are methods that process user requests, defined by frameworks. Furthermore, dependency injection and dynamic dispatching mechanisms in frameworks can lead to missing call relations, resulting in unsound and imprecise detection of SLVs. To address the above limitations, we enrich the call graph with the framework model by utilizing previous works [3, 38], completing entry points and missing call relations.

Forward Analysis:

$$IN_{entry} = true \quad (1)$$

$$IN_i = \bigvee_{p \in pred_i} (OUT_p) \quad (2)$$

$$OUT_i = \begin{cases} false & \text{session expiration} \\ IN_i & \text{otherwise} \end{cases} \quad (3)$$

Backward Analysis:

$$OUT_{exit} = true \quad (4)$$

$$OUT_i = \bigvee_{s \in succ_i} (IN_s) \quad (5)$$

$$IN_i = \begin{cases} false & \text{session expiration} \\ OUT_i & \text{otherwise} \end{cases} \quad (6)$$

$$\text{Result}_{\text{ASO}} = \text{IN}_{\text{ASO}}^{\text{forward}} \wedge \text{IN}_{\text{ASO}}^{\text{backward}} \quad (7)$$

Dataflow analysis. To detect SLVs, one possible approach is to traverse all paths in the ICFG from program entry to exit, identifying any paths where an ASO occurs without a session expiration. However, this method faces the potential issue of path explosion, making it computationally impractical for complex programs.

To address the issue of path explosion, we reformulate the control flow problem into a data flow problem, as illustrated in Equations 1-6. This transformation shifts the focus from identifying all paths leading to SLVs to determining the existence of at least one such path within each entry point.

We utilize the IFDS algorithm [49] to conduct the aforementioned data flow analysis. In the forward analysis phase, we initialize the data-flow value to *true* at the entry block of each entry point. This value is changed to *false* only when a session expiration is encountered. When multiple paths converge at a program point, their data-flow values are merged using the logical OR ($\vee$) operator. The backward analysis phase follows a similar analytical process.

The final results, as illustrated in Equation 7, demonstrate the merging of forward and backward analyses at the ASO point using a logical AND operation. If the merged outcome remains *true*, a SLV is reported.

Optimization. To further enhance the efficiency of SLVHound, we design an optimized approach. Our approach involves examining the set of reachable instructions from a given entry point, specifically looking for the co-occurrence of ASOs and session expiration. When both of them are present, we proceed with the dataflow analysis to identify potential SLVs. However, if only ASOs are detected without corresponding session expiration, we report a potential SLV without further analysis.

4 Evaluation

We implement SLVHound in WALA [26, 53], with 10K lines of Java code. The standard insensitive Andersen's analysis is employed to build the call graph and we extend WALA's IFDS framework for data-flow analysis. All control flow-related computations, including control dependencies and dominator relationships, are implemented using the built-in libraries of WALA. For the large language model, we utilize GPT-4o [43], accessed through OpenAI's API [42].

We evaluated SLVHound on 15 open-source Java Web applications, as detailed in Table 2. Our evaluation dataset comprises two groups: 8 applications with known vulnerabilities (rows 3-10) and 7 latest versions of popular open-source applications (rows 11-17). While the inclusion of applications with known vulnerabilities might appear to introduce bias, this selection was intentionally designed to validate our approach against documented vulnerabilities while maintaining the potential to discover new ones. For the second group, we employed a systematic selection process based on multiple criteria: use of Java programming language, popularity (measured by GitHub stars), diverse application categories, and implementation using common web frameworks (like Spring and Struts2, which are supported by existing static analysis tools [3, 38]). To ensure broad applicability, we deliberately included applications with diverse characteristics, notably four enterprise-level applications contributed by major technology companies: InLong (Tencent), Nacos (Alibaba), DolphinScheduler (DataSophon), and Graylog. The experiments were conducted on a MacBook Pro laptop equipped with a 4-core 2.2 GHz Intel Core i7 processor, 16 GB of memory, and MacOS Catalina 10.15.7.

Our evaluation aims to answer the following research questions:

- RQ1. How effective is SLVHound in inferring ASOs?
- RQ2. How effective is SLVHound in inferring session expiration?
- RQ3. How effective is SLVHound in detecting SLVs?
- RQ4. How efficient is SLVHound?
- RQ5. How does LLM-based login identification compare to rule-based approaches(ablation study)?

It is worth noting that we do not compare SLVHound with existing tools like SPACE [41] and SAWFIX [2] in our evaluation, as they focus on various session-related vulnerabilities but do not target SLVs. To the best of our knowledge, SLVHound is the first automated tool dedicated to SLV detection, and these vulnerabilities have traditionally only been identified through manual dynamic testing.

Table 2: Detailed Statistics of Evaluated Java Web Applications. This table provides a comprehensive overview of 15 Java web applications with the following metrics: application identification (Name), codebase statistics (Classes, Methods, Entries), authentication mechanism (Session Management), database structure information (Tables, Columns), project popularity (Stars), and functional categorization (Description).

Name	Application			Session	Database		Stars	Description
	Classes	Methods	Entries	Management	Tables	Columns		
SurveyKing-1.6 [27]	326	3,864	130	Custom	67	635	3.3k	Survey System
dolphinscheduler-3.2 [4]	2,195	18,633	373	Custom	34	298	12.7k	Data Management System
graylog2-server-5.0 [23]	6,763	45,448	434	Apache Shiro	15	93	7.4k	Log Management System
Apache inlong-1.6.0[5]	4,882	57,396	276	Apache Shiro	31	518	1.4k	Data Management System
shopizer-v3.0.1 [51]	1,203	9,039	374	Custom	81	465	3.5k	E-commerce Platform
MyHome-1.2 [29]	182	1,686	85	Custom	10	40	0.1k	Apartment Manage System
nacos-2.2 [1]	2,776	30,125	257	Custom	15	110	30.1k	Development Platform
Backend-2.6.0 [61]	72	481	27	Spring Security	3	19	0k	Educational Backend
CSKefu-7 [12]	855	11,236	403	Custom	121	2,932	2.7k	Customer Service System
xzs-3.8 [39]	198	1,710	81	Spring Security	13	114	3.1k	Exam System
Ruoyi-4.7.8 [62]	301	2,736	337	Apache Shiro	20	228	6k	Backend Management System
Roller-6.1.2 [17]	634	5,899	656	Spring Security	30	254	0.1k	Blog System
DBlog-2.3.4 [63]	325	2,092	173	Apache Shiro	22	234	5.1k	Blog System
Lin-cms-0.21 [58]	137	1,230	58	Custom	9	93	0.95k	Content Management System
PublicCMS-4.0 [50]	870	5,662	285	Custom	68	682	2k	Content Management System

Table 3: Results on detecting SLVs. True Positive (TP) represents correctly identified vulnerabilities, while False Positive (FP) indicates incorrectly flagged cases. The last 7 projects in this table do not have a false negative (FN) evaluation, represented as "N/A", because they were collected from popular open-source platforms that do not provide an FN evaluation benchmark. In contrast, the first 8 projects are known vulnerable projects and therefore underwent FN evaluation.

Application	# Login	# Filter	# Checkers	# Authentication Columns		# ASOs		# Session Expiration		# SLVs		
				TP	FP	TP	FP	TP(F/C)	FP	TP	FP	FN
SurveyKing-1.6	3	1	2	4	0	4	0	0/0	0	4	0	0
dolphinscheduler-3.2	2	2	4	4	0	3	0	0/3	0	2	0	0
graylog2-server-5.0	9	0	3	2	0	2	0	3/0	0	0	2	1
Apache inlong-1.6.0	3	7	6	2	0	3	0	1/0	0	2	0	0
shopizer-v3.0.1	6	2	4	4	0	8	0	0/0	0	8	0	0
MyHome-1.2	3	3	2	2	0	1	0	0/0	0	1	0	0
nacos-2.2	4	10	3	2	0	3	0	0/0	0	3	0	0
Backend-2.6.0	2	0	2	0	0	2	0	0/0	0	2	0	0
CSKefu-7	3	19	6	3	0	6	0	1/3	0	8	0	N/A
xzs-3.8	2	1	7	3	0	2	0	0/0	0	2	0	N/A
Ruoyi-4.7.8	4	3	2	4	0	2	0	3/0	0	2	0	N/A
Roller-6.1.2	1	12	3	3	0	3	0	0/0	0	3	0	N/A
DBlog-2.3.4	3	2	3	3	0	5	0	3/0	0	2	0	N/A
Lin-cms-0.21	2	2	3	3	0	3	0	0/0	0	3	0	N/A
PublicCMS-4.0	5	4	12	3	0	4	0	0/7	0	2	0	N/A
Total	52	68	62	42	0	51	0	11/13	0	44	2	1

4.1 RQ1. Identifying ASOs

Accurately identifying ASOs is essential for the effectiveness of SLVHound. Table 3 presents a comprehensive overview of the identified logins, filters, checkers, authentication columns, and the ASOs on columns 2–8.

SLVHound successfully identifies 52 login methods (Column 2) and 68 filters (Column 3). Notably, Column 2 reveals that almost all (14/15) examined applications implement multiple login methods. This result underscores a significant challenge again in the manual identification of login methods. Our approach, which leverages large language models to analyze method signatures and bodies, has proven highly effective in identifying these login methods comprehensively. While LLMs theoretically cannot guarantee complete absence of false positives, we have not observed any in our evaluation. This perfect precision can be attributed to developers generally following established conventions in implementing login functionality, aligning with the widespread standardization of authentication patterns in web development. Within these identified filters and login methods, SLVHound successfully recognizes 62 authentication checkers. Leveraging these checkers, SLVHound identifies 42 authentication columns that store authentication data such as passwords, email addresses, and phone numbers.

In the evaluation of various applications, we encountered a unique situation with application *Backend-2.6.0*. For this particular case, we were unable to identify specific authentication columns due to the use of lambda expressions in ORM operations. This use of lambda expressions poses a challenge for static analysis, as it renders the extraction of column information from the corresponding WALA IR (Intermediate Representation) infeasible. Despite this limitation, our approach was still able to locate the table containing the authentication columns. Thus, we adopted a conservative approach

Table 4: Results on detecting existing SLVs

Detected	CVE-2022-25590, CVE-2023-50270, CVE-2023-31065(2), CVE-2022-23063(5), MyHome-247, nacos-10095(3), backend-99
Not Detected	CVE-2023-41041

to ensure the soundness of SLVHound, treating any modifications to the located tables as ASOs.

Finally, SLVHound successfully identified 51 ASOs without any false positives, as confirmed through manual verification. It is worth noting that our evaluation focuses on true positives and false positives as the criteria for evaluating the ASOs identified by SLVHound. We do not evaluate false negatives because it is challenging to collect all ASOs across all applications.

4.2 RQ2. Identifying session expiration

Columns 9 and 10 in Table 3 present the details of session expiration. Column 9 displays session expiration identified within each web application, represented in the format *F/C*. Here, *F* denotes the number of framework API calls, while *C* represents the count of database DELETE operations related to session expiration. In total, SLVHound successfully identified 24 distinct session expirations. As shown in Column 10, our manual verification confirmed that no false session expiration was reported.

4.3 RQ3. Effectiveness of SLVHound

We evaluate the effectiveness of SLVHound in terms of its ability to detect existing and new vulnerabilities. As shown in Table 3, SLVHound reported a total of 46 (*TP*+*FP*) vulnerabilities, including 14 existing vulnerabilities and 32 new vulnerabilities. SLVHound reported a total of 24 vulnerabilities from projects with known vulnerabilities and 22 new vulnerabilities from new projects. Columns 11–13 in Table 3 summarize the results in detecting vulnerabilities.

4.3.1 Existing Vulnerabilities. SLVHound successfully identified 14 out of 15 existing SLVs across 8 Java applications, achieving a recall rate of 93.33%. The detected existing SLVs are shown in Table 4. The numbers in parentheses indicate that this vulnerability report involves multiple SLVs, meaning there are several paths where SLVs are present.

4.3.2 False Negatives. SLVHound failed to identify a true SLV in the *graylog2-server* application due to its frontend-implemented logout functionality, which fell outside our backend-focused analysis scope. The SLV occurred due to its frontend logout method lacked proper backend session expiration. It was later fixed by adding a backend session expiration request to the frontend logout method.

4.3.3 New Vulnerabilities. SLVHound reported a total of 32 new vulnerabilities, of which 30 have been confirmed as real vulnerabilities. Subsequently, we reported them to their corresponding maintainers, and 16 CVE IDs were granted.

4.3.4 False Positives. SLVHound reported two false positives for the *graylog2-server* application. The reason for these false positives stems from the application's use of the EventBus framework [20], which results in numerous indirect calls. WALA was unable to identify these indirect calls accurately. Consequently, it incorrectly determined that session expiration was not executed before or after ASOs, leading to the false reporting of SLVs.

Table 5: Ablation Study for Identifying Logins

Application	# Login	# Filter	# Checkers	# ASOs	
				TP	FP
SurveyKing-1.6	1	1	0	1	0
dolphinscheduler-3.2	2	2	4	3	0
graylog2-server-5.0	1	0	0	0	0
Apache inlong-1.6.0	1	7	0	1	0
shopizer-v3.0.1	2	2	4	8	0
MyHome-1.2	2	3	2	1	0
nacos-2.2	3	10	3	3	0
Backend-2.6.0	1	0	1	2	0
CSKefu-7	3	19	6	6	0
xzs-3.8	1	1	4	2	0
Ruoyi-4.7.8	4	3	0	1	0
Roller-6.1.2	2	12	0	0	0
DBlog-2.3.4	2	2	0	2	0
Lin-cms-0.21	2	2	3	3	0
PublicCMS-4.0	6	4	12	4	0
SUM	33	68	39	37	0

4.3.5 Case Studies. Here we discuss two interesting new SLVs identified by SLVHound.

Case-1. The *CSKefu* project implements two distinct URLs for user deletion: `POST /admin/user/delete {userId}`, `POST /api/user {delete, userId}`. In the backend, these URLs are handled by separate entry points, but both operations result in the deletion of the user from the same database table. Consequently, either request should trigger the expiration of the deleted user's session. Leveraging the advantages of static analysis, SLVHound was able to comprehensively analyze the application code, successfully identifying these two potential vulnerabilities.

Case-2. The *SurveyKing* project had previously reported a SLV, CVE-2022-25590, which was caused by sessions remaining valid and usable after user logout. Upon analyzing the patched version, SLVHound discovered that the vulnerability still persisted. We examined the patch and related issues and revealed that the developers had not properly expired the session, but merely cleared the session attribute values in the response upon logout. This approach fails to fundamentally address the SLV [28]. Consequently, we reported this issue again and received a new CVE.

4.4 RQ4. Efficiency of SLVHound

Figure 9 illustrates the time required to analyze the applications listed in Table 2. Among the 15 applications analyzed, the average analysis time is 155.54 seconds, with the longest analysis taking 580.75 seconds. The analysis process is divided into three main components: constructing the call graph, inferring authentication-sensitive operations and session expiration, and detecting vulnerabilities. For each application, the majority of the time is spent on constructing the call graph or inferring ASOs and session expiration. Notably, the time required for detecting vulnerabilities is comparatively minimal, which demonstrates the efficiency of our designed detection method.

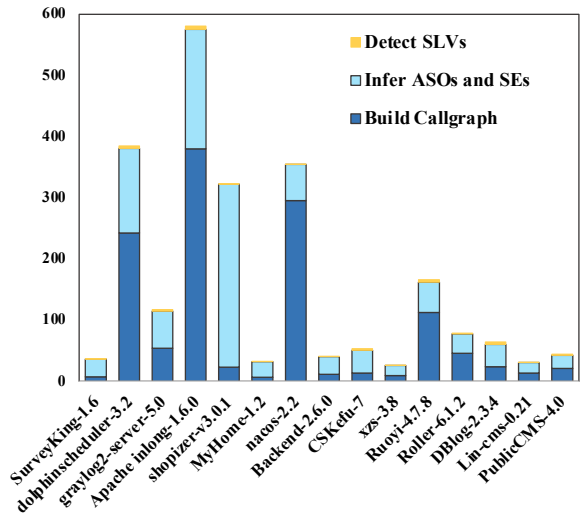


Figure 9: Analysis time (in seconds) of SLVHound

4.5 RQ5: Ablation Study

We perform an ablation study to assess the LLM component's contribution in login method identification, which serves as the foundation for subsequent ASO detection. We compared the LLM-based approach described in Section 3.1 and a more traditional rule-based approach, denoted as Login-LLM and Login-Rule respectively. The rule-based approach employs method name regex patterns(`(?i)\b\w*(?:login| signin|authenticate)\w*\b`) to identify login methods through pattern matching. This regex pattern was derived from a manual analysis of common login method naming conventions across our studied Web applications.

The results of both approaches are shown in Table 3 and Table 5. Login-LLM identified 52 login methods across all applications, while Login-Rule detected only 33 methods, achieving a 63.46% recall relative to Login-LLM. Login-LLM subsumes all rule-detected logins while discovering additional valid cases. This performance gap stems from LLMs' superior semantic understanding of code context, enabling more accurate login method recognition beyond simple pattern matching.

The impact of improved login identification cascades through the entire detection pipeline. With Login-LLM's more comprehensive login detection, our approach identified 62 checkers compared to 39 in the rule-based approach. This enhanced identification ultimately led to more thorough ASO detection, with Login-LLM identifying 51 ASOs compared to 37 by Login-Rule. This finding substantiates our methodological choice of adopting state-of-the-art LLM technology over conventional rule-based approaches, as it not only improves the initial login identification but also enables more complete downstream vulnerability detection.

5 Discussion

Token. Tokens are widely used to manage user identity information on the client side. Although it is typically assumed that tokens are not stored on the server, in practice, many Web applications implementing token mechanisms maintain expired tokens in a server-side database as a blacklist [48]. This approach enhances security by allowing the system to verify the validity of a token against this blacklist when a user accesses the application, thereby preventing SLVs. This verification process is analogous to traditional session management mechanisms. Therefore, this paper treats tokens as equivalent to sessions, without distinguishing between the two.

Filter chain. It is worth noting that certain Web applications, such as *graylog2-server-5.0* and *Backend-2.6.0* (as illustrated in Figure 3), do not employ the filter chain. Instead, these applications centralize their authentication checks within the login methods. Despite this difference, we still conceptualize this approach as part of the filter chain, albeit a simplified one where the chain consists solely of the login methods. This perspective allows us to maintain a consistent analytical framework across diverse authentication implementations.

Responsible Disclosure. All the new vulnerabilities detected by SLVHound can lead to serious consequences, including sensitive data leaks, data loss, and system compromise. Recognizing these potential risks, we took responsibility for disclosing all newly identified vulnerabilities in 15 Web applications through detailed reports. We contacted the respective organizations via their dedicated email addresses and security vulnerability reporting forms to report 30 new true vulnerabilities. Following industry-standard practices, we adhere to a 90-day responsible disclosure timeframe, which has been widely adopted by major security research organizations like Google's Project Zero [21]. We will refrain from publicly releasing any unresolved vulnerabilities until they have been addressed by the developers or until this disclosure period has elapsed.

Limitation. While SLVHound currently supports only Java Web applications (which consistently ranks among top 3 programming languages in TIOBE Index 2024 [52]) and has a strong enterprise presence, its design principles are adaptable to Web applications in other programming languages. Our approach focuses on language-agnostic session management patterns, with plans to extend to Python, PHP and Go. This adaptation would require supporting ORM and addressing incomplete call relations and missing entry points caused by Web frameworks.

For Web applications that do not use databases as backend storage, SLVHound may need adaptation to handle different storage mechanisms when identifying ASOs and session expiration. This limitation particularly affects applications using alternative storage solutions such as in-memory caches, file systems, or NoSQL databases.

As illustrated in Figure 4, Figure 5 and Figure 7, SLVHound presently requires manual configuration. However, it's important to note that this configuration is a one-time effort for each framework, as it defines framework-specific behaviors. Once established, these configurations can be reused across multiple applications utilizing the same frameworks, significantly reducing setup time for subsequent analyses. Moreover, thanks to large language models, writing these configurations has become more accessible. Even non-Web experts can quickly generate appropriate configurations by querying these models, further streamlining the setup process.Additionally,

our open-source implementation allows community-driven updates to these rules, ensuring they remain effective and up-to-date.

SLVHound also uses several heuristic rules, including those illustrated in Figure 8 and rules used to infer authentication checkers. While these heuristic rules may not be universally applicable across all applications, our experimental results demonstrate their effectiveness.

Our evaluation also revealed two key limitations of SLVHound. First, SLVHound's backend-focused analysis may miss vulnerabilities stemming from frontend session management, as shown in *graylog2-server*'s frontend logout implementation. Second, event-driven frameworks like EventBus can trigger false positives due to WALA's limitations in tracking indirect calls. We plan to enhance the call graph to address false positives and explore frontend code semantics analysis to reduce false negatives in future work.

6 Related Work

Empirical Study. Numerous studies [7, 24, 25, 30, 33] have examined vulnerabilities in Web application authentication mechanisms. These studies have comprehensively examined the prevalence, severity, and complexity of these vulnerabilities. However, most of this research has focused on well-known issues such as missing authentication, session fixation, session hijacking, weak passwords, and poorly configured session timeouts. In contrast, this paper presents the first comprehensive study specifically on SLVs.

Detection. Dynamic analysis provides distinct advantages in vulnerability detection through real-time monitoring of authentication state transitions and precise capture of session lifecycle events during runtime execution. Building on these capabilities, numerous studies [11, 13, 16, 34, 35, 35] have developed tools to identify authentication bypass vulnerabilities. Additionally, several research efforts [19, 37, 56, 57] have focused on vulnerabilities in session management, such as session fixation and session hijacking. These existing works employ dynamic analysis methods, simulating real-world attack scenarios and analyzing network traffic content to detect vulnerabilities.

Static analysis include tools like SPACE [41] and SAWFIX [2] that specialize in detecting distinct categories of authentication vulnerabilities. SPACE employs symbolic execution combined with bounded verification to systematically identify common security pattern violations, including missing authentication vulnerabilities, through rigorous evaluation against predefined security patterns. SAWFIX adopts abstract interpretation techniques to specifically address session fixation vulnerabilities, providing a formal framework for analyzing temporal dependencies in session management implementations. However, all these approaches are not designed to detect SLVs. To address this gap, this paper introduces SLVHound, a novel static analysis tool specifically designed to identify SLVs effectively.

7 Conclusion

This paper presents the first comprehensive study of Session Lingering Vulnerabilities (SLVs) in Web applications. Based on the findings from this study, we developed SLVHound, a novel detection tool that identifies SLVs by verifying the existence of execution paths that contain authentication-sensitive operations without corresponding session expiration. Our evaluation of SLVHound on 15 popular Web applications uncovered 46 potential vulnerabilities, of which 44 were confirmed as true SLVs, and 16 have been assigned CVE IDs.

Acknowledgments

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900) and the National Natural Science Foundation of China (62402474, 62132020 and 62202452).

References

[1] Alibaba. 2018. Nacos: An easy-to-use dynamic service discovery, configuration and service management platform for building cloud native applications. https://github.com/alibaba/nacos. Accessed: 2024-10-14.
[2] Abdelouahab Amira, Abdelraouf Ouadjaout, Abdelouahid Derhab, and Nadjib Badache. 2017. Sound and static analysis of session fixation vulnerabilities in php web applications. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 139–141.
[3] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 794–807.
[4] Apache DolphinScheduler Project. 2019. Apache DolphinScheduler. https://github.com/apache/dolphinscheduler. Accessed: 2024-10-15.
[5] Apache InLong Project. 2019. Apache InLong. https://github.com/apache/inlong. Accessed: 2024-10-15.
[6] Michael Burrows, Martin Abadi, and Roger Needham. 1990. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)* 8, 1 (1990), 18–36.
[7] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. 2017. Surviving the web: A journey into web session security. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–34.
[8] Common Weakness Enumeration. 2023. CWE-613: Insufficient Session Expiration. Web Page. https://cwe.mitre.org/data/definitions/613.html [Online; accessed: Access Date].
[9] Hibernate Community. 2024. *Hibernate - ORM framework*. https://hibernate.org/ Accessed: 2024-10-14.
[10] MyBatis Community. 2024. MyBatis - Persistence framework. https://mybatis.org/mybatis-3/. Accessed: 2024-10-14.
[11] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. 2007. Swaddler: An Approach for the Anomaly-Based Detection of State Violations in Web Applications. In *Recent Advances in Intrusion Detection*, Christopher Kruegel, Richard Lippmann, and Andrew Clark (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–86.
[12] cskefu. 2018. cskefu. Accessed online. https://github.com/cskefu/cskefu [Online; accessed August 12, 2025].
[13] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, USA, 267–282.
[14] Endgrate. 2024. *10 Session Management Security Best Practices*. https://endgrate.com/blog/10-session-management-security-best-practices
[15] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. 2024. Large language models for code analysis: Do {LLMs} really do their job?. In *33rd USENIX Security Symposium (USENIX Security 24)*. 829–846.
[16] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2010. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, USA, 10.
[17] Apache Software Foundation. 2005. Roller: Open Source Java Blog Software. https://github.com/apache/roller. Accessed: 2024-10-14.
[18] Eclipse Foundation. 2024. *Jakarta Persistence - Specification*. https://jakarta.ee/specifications/persistence/ Accessed: 2024-10-14.
[19] Nasrin Garmabi and Mohammad Ali Hadavi. 2021. Automatic Detection and Risk Assessment of Session Management Vulnerabilities in Web Applications. In *2021 11th International Conference on Computer Engineering and Knowledge (ICCKE)*. IEEE, 41–47.
[20] Google. 2024. Guava: Google Core Libraries for Java. https://github.com/google/guava/wiki/EventBusExplained Accessed: 2024-10-15.

[21] Google Project Zero. 2021. Project Zero: Vulnerability Disclosure FAQ. https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-faq.html. Accessed: 2024.

[22] Paul A Grassi, Elaine M Newton, Ray A Perlner, Andrew R Regenscheid, William E Burr, Justin P Richer, Naomi B Lefkovitz, Jamie M Danker, Yee-Yin Choong, Kristen Greene, et al. 2017. Digital identity guidelines: authentication and lifecycle management. (2017).

[23] Graylog2. 2010. Graylog2 Server. https://github.com/Graylog2/graylog2-server.

[24] Md. Maruf Hassan, Shamima Sultana Nipa, Marjana Akter, Rafita Haque, Fabiha Nawar Deepa, Mostafijur Rahman, Mohd. Shadab Siddiqui, and Md. Hasan Sharif. 2018. Broken Authentication and Session Management Vulnerability: A Case Study of Web Application. *International journal of simulation: systems, science and technology* (2018). https://api.semanticscholar.org/CorpusID:68089883

[25] Daniel Huluka and Oliver Popov. 2012. Root cause analysis of session management and broken authentication vulnerabilities. In *World Congress on Internet Security (WorldCIS-2012)*. 82–86.

[26] IBM. 2006. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/

[27] javahuang. 2005. SurveyKing: Make a better survey system. GitHub repository. https://github.com/javahuang/SurveyKing

[28] javahuang. 2023. Issue #7: There is a logout logic vulnerability in the background. https://github.com/javahuang/SurveyKing/issues/7. Accessed: 2024-10-12.

[29] JMPrathab. 2020. MyHome: A Smart Home Management System. https://github.com/jmprathab/MyHome. Accessed: 2024-10-14.

[30] Manuel Karl, Marius Musch, Guoli Ma, Martin Johns, and Sebastian Lekies. 2022. No keys to the kingdom required: a comprehensive investigation of missing authentication vulnerabilities in the wild. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 619–632. https://doi.org/10.1145/3517745.3561446

[31] Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. 2024. LSAST–Enhancing Cybersecurity through LLM-supported Static Application Security Testing. *arXiv preprint arXiv:2409.15735* (2024).

[32] Redis Labs. 2024. *Redis - In-memory data structure store.* https://redis.io/ Accessed: 2024-10-14.

[33] Yuriy Lakh, Elena Nyemkova, Andrian Piskozub, and Viktor Yanishevskyi. 2021. Investigation of the Broken Authentication Vulnerability in Web Applications. In *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 2. 928–931. https://doi.org/10.1109/IDAACS53288.2021.9660889

[34] Xiaowei Li and Yuan Xue. 2011. BLOCK: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 247–256. https://doi.org/10.1145/2076732.2076767

[35] Xiaowei Li, Wei Yan, and Yuan Xue. 2012. SENTINEL: securing database from logic flaws in web applications. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY '12)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2133601.2133605

[36] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[37] Raymond Lukanta, Yudistira Asnar, and A Imam Kistijantoro. 2014. A vulnerability scanning tool for session management vulnerabilities. In *2014 International conference on data and software engineering (ICODSE)*. IEEE, 1–6.

[38] Haining Meng, Haofeng Li, Jie Lu, Chenghang Shi, Liqing Cao, Lian Li, and Lin Gao. 2025. AutoWeb: Automatically Inferring Web Framework Semantics via Configuration Mutation. In *Engineering of Complex Computer Systems*. Springer Nature Switzerland, Cham, 369–389.

[39] Mindskip. 2019. XZS-MySQL: A repository for MySQL related projects and resources. https://github.com/mindskip/xzs-mysql. Accessed: 2024-10-14.

[40] Inc. MongoDB. 2024. *MongoDB - The database for modern applications.* https://www.mongodb.com/ Accessed: 2024-10-14.

[41] Joseph P Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering*. 947–958.

[42] OpenAI. 2023. OpenAI API. https://platform.openai.com/docs Accessed: 2024-10-15.

[43] OpenAI. 2024. GPT-4o: An Optimized Version of GPT-4. https://openai.com/index/hello-gpt-4o Accessed: 2024-10-15.

[44] owasp. 2024. Cross Site Request Forgery (CSRF). Web Page. https://owasp.org/www-community/attacks/csrf [Online; accessed: Access Date].

[45] owasp. 2024. Cross Site Scripting (XSS). Web Page. https://owasp.org/www-community/attacks/xss/ [Online; accessed: Access Date].

[46] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *CoRR* (2023).

[47] Inc. Pivotal Software. 2009. Spring Security. GitHub repository. https://github.com/spring-projects/spring-security

[48] Rakan. 2024. JWT can fit as an authentication system with a blacklist technique. Web Page. https://dev.to/irakan/jwt-can-fit-as-an-authentication-system-with-a-blacklist-technique-4ohl [Online; accessed: Access Date].

[49] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[50] sanluan. 2016. PublicCMS: An open-source content management system. GitHub repository. https://github.com/sanluan/PublicCMS

[51] Shopizer. 2013. Shopizer: Open Source e-commerce Software. https://github.com/shopizer-ecommerce/shopizer. Accessed: 2024-10-14.

[52] TIOBE Software. 2024. TIOBE Programming Community Index. https://www.tiobe.com/tiobe-index/. Accessed: January 2024.

[53] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 196–232.

[54] Security StackExchange. 2015. *How long should a session absolute timeout be?* https://security.stackexchange.com/questions/106786/how-long-should-a-session-absolute-timeout-be

[55] Security StackExchange. 2021. *How do big websites have practically infinite session duration?* https://security.stackexchange.com/questions/256438/how-do-big-websites-have-practically-infinite-session-duration

[56] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. 2010. Automated detection of session fixation vulnerabilities. In *Proceedings of the 19th international conference on World wide web*. 1191–1192.

[57] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. 2012. Automated detection of session management vulnerabilities in web applications. In *2012 Tenth Annual International Conference on Privacy, Security and Trust*. IEEE, 112–119.

[58] TaleLin. 2019. Lin-CMS-Spring-Boot: A simple and practical CMS implemented with Spring Boot. https://github.com/TaleLin/lin-cms-spring-boot. Accessed: 2024-10-14.

[59] The Spring Team. 2024. *Spring Data JPA - Reference Documentation.* https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[60] Steve Vinoski. 2002. Chain of responsibility. *IEEE Internet Computing* 6, 6 (2002), 80–83.

[61] Vladyslav-Team. 2023. Backend: A repository for backend development. https://github.com/Vladyslav-Team/backend. Accessed: 2024-10-14.

[62] yangzongzhuan. 2018. RuoYi. https://github.com/yangzongzhuan/RuoYi. Accessed: 2024-10-14.

[63] Yadong Zhang. 2017. DBlog: A simple, fast and powerful blog system based on Spring Boot. https://gitee.com/yadong.zhang/DBlog. Accessed: 2024-10-14.