AutoMutSec: Mutation Testing for Authorization Policies in Spring Security

Caio Jhonatan Alves Pereira
Federal University of Campina Grande
Brazil
caio.jhonatan.pereira@ccc.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Brazil
everton@computacao.ufcg.edu.br

Abstract

Authorization failures remain one of the most critical security risks in modern web applications, yet the effectiveness of authorization testing is rarely assessed. In Spring Security, authorization policies are implemented through annotations and configuration constructs that fall outside the scope of conventional mutation operators, which primarily target language-level faults. As a result, test suites can achieve high mutation scores, while access-control behavior remains largely untested. This paper presents AutoMutSec, a mutation testing tool that introduces ten authorization-specific mutation operators for Spring Security applications. We evaluated AutoMutSec on five real-world Spring Boot systems and compared its results with those produced by PIT, the standard mutation testing tool for Java. AutoMutSec generated 1,745 mutants across the evaluated systems, yielding mutation scores ranging from 0% to 82.42%. We found no consistent relationship between AutoMutSec scores and traditional testing metrics such as PIT mutation scores or code coverage. Notably, some systems with PIT scores above 50% achieved AutoMutSec scores of 0%, revealing that authorization logic can remain entirely untested despite otherwise strong testing indicators. These results suggest that authorization testing effectiveness and general testing effectiveness may be largely independent dimensions. The additional execution overhead remained practical in most systems. Overall, our findings indicate that traditional mutation testing is insufficient to assess authorization testing adequacy and that authorization-specific mutation operators provide complementary insights into security testing effectiveness.

Keywords

Mutation Testing, Spring Security, Authorization Testing, Security Testing

1 Introduction

Web-based systems often rely on authorization mechanisms to restrict access to protected resources. However, these mechanisms are frequently implemented incorrectly, leading to security vulnerabilities that may expose sensitive functionality or data [15]. Reflecting the severity of this problem, Broken Access Control has been ranked as the most critical security risk in the OWASP Top 10[1]. Frameworks such as Spring Security are widely adopted in industry [10] to support the implementation of authorization policies, making their correct usage and the detection of misconfigurations relevant concerns for software security research.

Ensuring that authorization policies are correctly enforced is particularly challenging because violations are often subtle and difficult to detect. A single misconfigured rule or overlooked endpoint may grant unauthorized access to sensitive information, potentially violating regulations such as LGPD and GDPR [4, 6]. Although testing is widely recognized as a means of increasing confidence in system correctness and security [11, 14], authorization requirements remain frequently undertested in practice. Recent evidence suggests that authorization testing consumes a substantial portion of the effort involved in web penetration testing and remains one of the few security-testing activities for which automation is still limited [7].

This challenge is particularly evident in Spring Security applications, where authorization policies are commonly specified through annotations and configuration constructs rather than explicit business logic. As a result, the correctness of access-control rules is often neglected in conventional unit and integration testing approaches.

Even applications with extensive automated test suites may leave authorization behavior largely unverified. To illustrate this issue, consider the test suite of the Gymstock project[2]: although the application defines several role-based authorization rules at the controller layer, the tests focus exclusively on service-layer functionality, leaving access-control behaviors untested. This pattern recurs across projects where security configuration and application logic are tested independently. Detecting such gaps typically requires labor-intensive manual inspection of endpoints and security configurations across multiple misconfiguration scenarios, a process that becomes increasingly difficult as systems evolve and grow in size. These scenarios highlight the need for automated techniques capable of exposing authorization-related weaknesses that traditional testing approaches may overlook.

A fundamental challenge in this context is to determine whether a test suite is actually capable of detecting authorization failures rather than simply exercising secured functionality. Mutation testing addresses this challenge by introducing artificial faults into the source code, known as mutants, and evaluating whether the test suite can detect them [12]. When a mutant represents an authorization defect and survives execution, it reveals a concrete gap in the test suite's ability to detect security-related failures. Consequently, mutation testing is particularly well suited for assessing the effectiveness of authorization tests in Spring Security applications.

However, existing mutation testing tools provide limited support for evaluating authorization mechanisms implemented with Spring Security. Traditional mutation operators focus on language-level constructs, such as arithmetic expressions, relational operators, and control-flow statements. In contrast, Spring Security policies are

[1] https://owasp.org/Top10/2025/

[2] https://github.com/asafeorneles/gymstock

often defined through annotations and configuration expressions that fall outside the scope of conventional mutation operators. Consequently, a test suite may achieve a high mutation score while still failing to detect critical access-control defects. This creates an important gap in the assessment of security testing effectiveness for Spring Security applications.

To address this gap, we propose AutoMutSec, a mutation testing tool that introduces ten authorization-specific mutation operators targeting Spring Security configurations. We evaluate AutoMutSec through an empirical study involving real-world Spring applications. Our study investigates whether authorization-specific mutants can reveal weaknesses in existing test suites, examines the additional insights provided by security-oriented mutation operators compared with traditional mutation testing approaches, and assesses the practical cost of applying AutoMutSec in real development settings.

This paper makes the following contributions:

- **Ten authorization-specific mutation operators for Spring Security.** Based on common authorization misconfigurations in Spring Security applications, we derive ten mutation operators that emulate realistic access-control faults.
- **AutoMutSec, a mutation testing tool for authorization testing.** We design and implement AutoMutSec, a mutation testing tool capable of automatically generating and executing authorization-specific mutants targeting Spring Security configurations.
- **An empirical evaluation on real-world Spring applications.** We conduct a study on five Spring Boot projects (four open-source and a private system) to investigate the extent to which authorization-specific mutants reveal weaknesses that remain undetected by existing test suites.
- **Comparative evaluation of AutoMutSec and PIT in terms of fault detection and execution cost.** We compare the insights provided by AutoMutSec against those obtained from traditional mutation testing approaches and assess the execution cost of applying authorization-specific mutation analysis in practice.

2 Motivational Example

Consider a bug reported in the SpringUserFramework repository[3], a Spring Boot-based user management framework built on Spring Security. The issue, titled "AuthorityService doesn't include role names as granted authorities, breaking @PreAuthorize("hasRole(...)")" (Issue #293)[4], describes a runtime failure in `UserEmailService.initiateAdminPasswordReset()`, resulting in an HTTP 500 error. The root cause lies in `AuthorityService.getAuthoritiesFromRoles()`, which maps roles and privileges into Spring Security granted authorities. In the reported implementation, only privileges such as `ADMIN_PRIVILEGE` and `LOGIN_PRIVILEGE` are included, while role-based authorities such as `ROLE_ADMIN` are omitted, as shown in Listing 1.

Listing 1: Misconfigured role-to-authority mapping in AuthorityService.

```
roles.stream()
    .flatMap(role -> role.getPrivileges().stream())  // only
     privileges, roles omitted
    .map(Privilege::getName)
    .map(SimpleGrantedAuthority::new)
    .collect(toSet());
```

As a consequence, authorization checks that depend on role names, such as `@PreAuthorize("hasRole('ADMIN')")`, silently fail at runtime, even when the annotation is correctly declared. The repository's existing test suite includes the test shown in Listing 2 for this method. This test only verifies the presence and content of the annotation through reflection, without executing the method under a real Spring Security context. Because `@PreAuthorize` depends on Spring Security's runtime infrastructure for authentication and authority resolution, the actual authorization logic is never evaluated and the role-to-authority mapping failure goes undetected.

Listing 2: Existing test verifying the presence of the @PreAuthorize annotation via reflection.

```
@Test
@DisplayName("initiateAdminPasswordReset(User, String, boolean)
     has @PreAuthorize annotation")
void initiateAdminPasswordReset_hasPreAuthorizeAnnotation()
     throws NoSuchMethodException {
    Method method = UserEmailService.class.getMethod(
        "initiateAdminPasswordReset",
        User.class,
        String.class, boolean.class
    );
    PreAuthorize annotation =
      method.getAnnotation(PreAuthorize.class);
    assertThat(annotation).isNotNull();
    assertThat(annotation.value())
            .isEqualTo("hasRole('ADMIN')");
}
```

A more effective test would execute the method within a properly configured security context, as shown in Listing 3[5]. This test would have detected the misconfiguration described in the issue.

Listing 3: Improved test executing the method under a real Spring Security context.

```
@Test
@WithMockUser(roles = "ADMIN")
void initiateAdminPasswordReset_shouldExecute_
     whenUserHasAdminRole() {
    User user = new User();
    assertDoesNotThrow(() ->
        userEmailService.initiateAdminPasswordReset(user, "
     test@test.com", true)
); }
```

However, it still covers only the positive case for a single role. A robust authorization test suite must also verify that unauthorized roles are correctly denied, exercising the full range of access-control scenarios. Systematically identifying these gaps across endpoints and role combinations is precisely the problem that AutoMutSec is designed to address.

3 Background

Mutation testing is a fault-based testing technique that evaluates test suite effectiveness by introducing artificial faults into the source

[3]https://github.com/devondragon/SpringUserFramework/
[4]https://github.com/devondragon/SpringUserFramework/issues/293

[5]A mutant generated by AutoMutSec that removes the `ROLE_ADMIN` authority from the security context would survive under the test in Listing 2 but be killed by a test such as the one in Listing 3, motivating its inclusion in the test suite.

code [3]. Each modified version of the program is called a mutant. A mutant is said to be *killed* if at least one test fails after the modification is applied, otherwise, it *survives*, indicating that the test suite failed to distinguish the faulty version from the original.

For example, a conditional expression such as `if (a > b)` may be mutated to `if (a >= b)`. If no test detects this change, the mutant survives, suggesting either insufficient coverage or weak assertions.

The mutation score is defined as the ratio of killed mutants to the total number of generated mutants. It quantifies how effectively a test suite detects injected faults. A low mutation score indicates that many faults go undetected, pointing to gaps in test adequacy. In the context of this work, this metric is used to assess whether existing test suites are capable of detecting authorization-specific faults in Spring Security configurations.

Several mutation testing tools have been proposed for Java applications. PIT [5] is one of the most widely adopted, offering operators targeting language-level constructs such as conditional expressions, arithmetic operators, return values, and method calls.

3.1 Security Issues

Web applications that require data integrity frequently depend on authentication and authorization mechanisms to enforce access control policies [1]. Implementation flaws in these mechanisms can introduce vulnerabilities that grant unauthorized access to protected resources and sensitive information.

Real-world vulnerabilities in Spring Security authorization have been documented in recent CVEs. CVE-2025-41248[6] describes an authorization bypass caused by incorrect resolution of method security annotations such as `@PreAuthorize` in generic type hierarchies. Listing 4 illustrates this scenario.

Listing 4: Authorization bypass due to incorrect annotation resolution in generic type hierarchies (CVE-2025-41248).

```
public interface UserService<T> {
    @PreAuthorize("hasRole('ADMIN')")
    T findById(Long id);
    }
@Service
public class CustomerService implements UserService<Customer> {
    @Override
    public Customer findById(Long id) {
        return repository.findById(id);
    } }
```

Access to this method should be restricted to users with the `ADMIN` role. However, due to incorrect annotation resolution in generic type hierarchies, the security constraint may not be enforced, allowing unauthorized users to invoke the method. A related vulnerability, CVE-2025-41232[7], allowed protected private methods to be invoked without proper authorization checks due to incorrect detection of method security annotations.

These cases share a common characteristic: the faults are not detectable at compile time, remain silent until triggered by specific runtime conditions or exploited in production. This makes authorization defects particularly suitable targets for mutation testing, a technique that injects representative faults into the code and evaluates whether the test suite is capable of detecting them before they reach production.

3.2 Java Spring Security

Spring Boot is a framework for developing Java-based applications and is commonly adopted in enterprise software projects [10]. Spring Security is a security framework that provides built-in mechanisms for implementing authentication and authorization in Spring-based applications[8]. Although these mechanisms simplify the development of secure applications, vulnerabilities may still arise from incorrect configurations or improper use of the framework [8].

When using Spring Security annotations, developers must ensure that roles, permissions, and authorities are correctly defined and applied. This task is more error-prone than it may initially appear. Incorrect role names, mismatched authority identifiers, or misuse of logical operators can introduce subtle authorization faults that do not produce exceptions but silently affect runtime behavior.

A common example is shown in Listings 5 and 6: a method intended to be accessible only to administrators is protected with `@PreAuthorize("hasRole('ADMINISTRATOR')")` while authenticated users are assigned the role `ADMIN`. In Spring Security, `hasRole(X)` is evaluated against the authority `ROLE_X`, so `hasRole("ADMINISTRATOR")` requires `ROLE_ADMINISTRATOR`, but legitimate administrators hold `ROLE_ADMIN`. As a result, access is denied for all users, including those with administrative privileges, without any runtime exception.

Listing 5: Method protected with an incorrect role name.

```
@PreAuthorize("hasRole('ADMINISTRATOR')")
public String manageStaffs() { ... }
```

Listing 6: User entity with ADMIN role assigned as granted authority.

```
class User implements UserDetails {
    private Role role; // Role.ADMIN
    @Override
    public Collection<? extends GrantedAuthority>
      getAuthorities() {
    return List.of(new SimpleGrantedAuthority("ROLE_" + role));
    } }
```

Spring Security provides several built-in authorization operators. The operators `hasRole` and `hasAuthority` grant access only if the authenticated user possesses a specific role or authority, respectively; `hasAnyRole` and `hasAnyAuthority` grant access when the user holds at least one element from a predefined set. The `hasPermission` operator delegates the authorization decision to a custom `PermissionEvaluator`, enabling object-level access control based on application-specific business logic. Spring Security also supports custom authorization expressions that delegate to application-defined components (e.g., `@PreAuthorize("@securityService.isOwner(#id)")`). Unlike built-in operators, custom expressions depend on application-specific logic with arbitrary names and semantics, making them harder to target with generic mutation operators. AutoMutSec therefore focuses on built-in authorization

[6] https://www.cve.org/CVERecord?id=CVE-2025-41248
[7] https://www.cve.org/CVERecord?id=CVE-2025-41232
[8] https://docs.spring.io/spring-security/reference/

expressions, where the semantics are well-defined and representative mutation operators can be systematically derived.

4 Mutation Operators

Traditional mutation testing operators are primarily designed to mutate conventional source code constructs, such as statements, expressions, and control-flow elements. However, authorization rules in Spring Security are frequently specified as string-based expressions embedded within annotations such as `@PreAuthorize`. Conventional mutation operators do not target these constructs, leaving a gap in the assessment of authorization test adequacy.

To address this gap, we propose ten mutation operators specifically designed for authorization expressions in Spring Security. Each operator models a realistic misconfiguration that developers may inadvertently introduce when defining access-control policies. The operators were derived from two complementary sources: (i) documented authorization vulnerabilities in Spring Security applications (e.g., CVE-2025-41248 and issues 293, discussed in Sections 2 and 3.1), and (ii) the authors' practical experience developing and reviewing Spring Security configurations, which revealed recurring misconfiguration patterns such as incorrect logical connectors, incorrect identifiers, and missing negations. Table 1 summarizes the proposed operators.

4.1 Logical Security Operator Replacement (LSOR)

Authorization expressions in Spring Security may combine multiple conditions using logical operators such as `AND` and `OR`. Swapping these operators changes the access semantics significantly: an expression that previously required all conditions to hold now requires only one, or vice versa. This emulates a common mistake where developers select the wrong logical connector when composing access-control rules, letting a user with only one of the required credentials gain unauthorized access undetected.
Transformation: $AND \leftrightarrow OR$

Listing 7: LSOR: replacing AND with OR.

```
// Original
hasRole('ADMIN') AND hasAuthority('WRITE')
// Mutant
hasRole('ADMIN') OR hasAuthority('WRITE')
```

4.2 Logical Negation Security Operator (LNSO)

Inserting a negation operator inverts the access decision for a given expression, granting access to users who should be denied and denying access to users who should be granted. This emulates scenarios where a developer accidentally negates a condition or applies negation to the wrong subexpression, going undetected whenever the test suite never checks both that legitimate users are still granted access and that users lacking the negated condition are correctly denied it.
Transformation: $E \rightarrow !E$, where E is an authorization expression.

Listing 8: LNSO: inserting negation into an authorization expression.

```
// Original
hasRole('ADMIN') AND hasAuthority('WRITE')
// Mutants
!hasRole('ADMIN') AND hasAuthority('WRITE')
hasRole('ADMIN') AND !hasAuthority('WRITE')
```

4.3 Invalid Security Identifier Replacement (ISIR)

This operator replaces a valid role or authority identifier with a value that does not correspond to any identifier defined in the application. As a result, the authorization expression can never be satisfied, effectively denying access to all users. This emulates typos or copy-paste errors in identifier names, going undetected whenever the test suite never verifies that a user with the correct role can actually access the resource.
Transformation: $Identifier \rightarrow InvalidIdentifier$

Listing 9: ISIR: replacing a valid identifier with an invalid one.

```
// Original
hasRole('ADMIN')
// Mutant
hasRole('INVALID_ROLE')
```

4.4 Authorization Replacement Operator (ARO)

This operator replaces a valid identifier with a different valid identifier of the same type. Unlike ISIR, the resulting expression is semantically valid but grants access to a different set of users than intended. This emulates mistakes where a developer references the wrong role or authority from the application's defined set — an error that survives whenever the test suite never confirms that access is granted to the correct identifier and denied to other valid ones.
Transformation: $Identifier_A \rightarrow Identifier_B$, where both belong to the same category.

Listing 10: ARO: replacing a valid identifier with another valid identifier.

```
// Original
hasRole('ADMIN')
// Mutant
hasRole('MANAGER')
```

4.5 Security Expression Type Replacement (SETR)

This operator replaces the authorization function while keeping the identifier unchanged. In Spring Security, `hasRole(X)` checks for the authority `ROLE_X`, while `hasAuthority(X)` checks for the authority `X` exactly. Swapping these functions changes the underlying authority resolution mechanism and may cause authorization to fail or pass incorrectly, depending on how authorities are assigned to users — a mismatch that survives whenever the test suite never checks which resolution mechanism is actually applied.
Transformation: $hasRole(X) \leftrightarrow hasAuthority(X)$

Listing 11: SETR: replacing the authorization function type.

```
// Original
hasRole('ADMIN')
// Mutant
hasAuthority('ADMIN')
```

Table 1: Proposed authorization-specific mutation operators.

Acronym	Name	Transformation
LSOR	Logical Security Operator Replacement	$AND \leftrightarrow OR$
LNSO	Logical Negation Security Operator	$E \rightarrow !E$
ISIR	Invalid Security Identifier Replacement	$Id \rightarrow InvalidId$
ARO	Authorization Replacement Operator	$Id_A \rightarrow Id_B$
SETR	Security Expression Type Replacement	$hasRole(X) \leftrightarrow hasAuthority(X)$
SITR	Security Identifier Type Replacement	$RoleId \leftrightarrow AuthorityId$
PARO	Permit All Replacement Operator	$E \rightarrow permitAll()$
DARO	Deny All Replacement Operator	$E \rightarrow denyAll()$
APRO	Authorization Parameter Reduction Operator	$F(p_1, ..., p_n) \rightarrow F(p_1, ..., p_{n-1})$
EAAO	Empty Authorization Argument Operator	$F(p_1, ..., p_n) \rightarrow F('')$

4.6 Security Identifier Type Replacement (SITR)

While SETR swaps the authorization function, SITR keeps the function and replaces the identifier with one belonging to a different category. For instance, substituting a role identifier inside `hasRole` with an authority identifier, or vice versa. This emulates a mistake where a developer uses an identifier from the wrong category within a given authorization function, a mismatch that goes undetected whenever the test suite never validates that identifiers correspond to the correct category.

Transformation: $RoleIdentifier \leftrightarrow AuthorityIdentifier$, within the same operator.

Listing 12: SITR: replacing an identifier with one from a different category.

```
// Original
hasRole('ADMIN')       // ADMIN is a role identifier
// Mutant
hasRole('WRITE')       // WRITE is an authority identifier
```

4.7 Permit All Replacement Operator (PARO)

This operator replaces an authorization expression with `permitAll()`, removing all access restrictions from the annotated method. This emulates a scenario where a developer accidentally disables access control, either by misconfiguring a security rule or by leaving a development-time bypass in production code.

Transformation: $E \rightarrow permitAll()$, where E is an authorization expression.

Listing 13: PARO: replacing an authorization expression with permitAll().

```
// Original
hasRole('ADMIN') AND hasAuthority('WRITE')
// Mutant
permitAll()
```

A surviving PARO mutant is particularly critical: it indicates that the test suite does not verify access restriction at all, meaning that any user (including unauthenticated ones) could access the protected resource without being detected by the tests.

4.8 Deny All Replacement Operator (DARO)

This operator replaces an authorization expression with `denyAll()`, blocking access for all users regardless of their credentials. This emulates an overly restrictive misconfiguration that prevents legitimate users from accessing resources they should be able to reach.

Transformation: $E \rightarrow denyAll()$, where E is an authorization expression.

Listing 14: DARO: replacing an authorization expression with denyAll().

```
// Original
hasRole('ADMIN') AND hasAuthority('WRITE')
// Mutant
denyAll()
```

A surviving DARO mutant indicates that the test suite does not include positive authorization cases — that is, it never verifies that a legitimate user can successfully access the protected method, leaving overly restrictive misconfigurations undetected.

4.9 Authorization Parameter Reduction Operator (APRO)

Functions such as `hasAnyRole` and `hasAnyAuthority` accept multiple authorization parameters, granting access when the authenticated user satisfies at least one. Removing one or more parameters from such expressions narrows the set of authorized users, potentially denying access to users who should be permitted. This emulates a common mistake where a developer omits a role or authority during maintenance — for example, when refactoring a list of allowed roles — going undetected whenever the test suite never checks access for the removed identifier.

APRO generates one mutant for each parameter that can be individually removed from the expression. For an expression with n parameters, APRO produces n mutants, one for each position.

Transformation: $F(p_1, p_2, ..., p_n) \rightarrow F(p_1, ..., p_{n-1})$, where F is an authorization function accepting multiple parameters and p_i denotes an individual authorization parameter.

Listing 15: APRO: removing an authorization parameter from a multi-argument expression.

```
// Original
hasAnyRole('EDITOR', 'ADMIN')
// Mutants
hasAnyRole('ADMIN')
hasAnyRole('EDITOR')
```

4.10 Empty Authorization Argument Operator (EAAO)

Replacing authorization parameters with an empty string produces an expression that can never be satisfied by any valid user, since no user holds an empty role or authority. This emulates scenarios where a developer accidentally clears an authorization argument (e.g., through a misconfigured template, a broken import, or an incomplete refactoring that leaves an empty string in place of a role name).

Although EAAO and DARO produce similar observable effects (both deny access to all users) they represent distinct fault patterns. DARO emulates an explicit access block, while EAAO emulates a subtler fault where the expression structure is preserved but the authorization arguments are accidentally cleared. A test suite that kills DARO mutants but not EAAO mutants detects explicit blocking but not argument-level corruption.

Transformation: $F(p_1, p_2, ..., p_n) \rightarrow F('')$, where F is an authorization function and p_i represents an authorization parameter.

Listing 16: EAAO: replacing authorization arguments with an empty string.

```
// Original
hasAnyRole('EDITOR', 'ADMIN')
// Mutant
hasAnyRole('')
```

A surviving EAAO mutant indicates that the test suite does not include positive authorization cases for the affected method — no test verifies that a user with a valid role can access the resource, leaving empty-argument errors undetected. This is similar in effect to DARO, but targets a more specific and realistic fault pattern.

5 AutoMutSec

AutoMutSec is a mutation testing tool designed to evaluate the effectiveness of test suites with respect to authorization mechanisms in Spring Security applications. The tool takes a Spring Boot project as input, applies the authorization-specific mutation operators described in Section 4, executes the test suite against each generated mutant, and produces a report summarizing which mutants were killed and which survived. AutoMutSec is publicly available at [9]

5.1 Architecture Overview

Figure 1 presents an overview of AutoMutSec's architecture. The main execution flow is controlled by the `CLIController` class, which acts as the entry point of the application. Before applying any mutations, AutoMutSec copies the target project to a temporary directory, preserving the original source code throughout the process.

The `DirectoryScan` component recursively scans the project directory structure and collects all Java source files. These files are then passed to the `CodeAnalyzer`, which identifies mutation points in authorization expressions using regular expressions and heuristics based on Spring Security annotation patterns. For each identified point, the component extracts the class name, method name, line number, and authorization expression value.

[9] https://github.com/DovCaio/mutate_security_configs

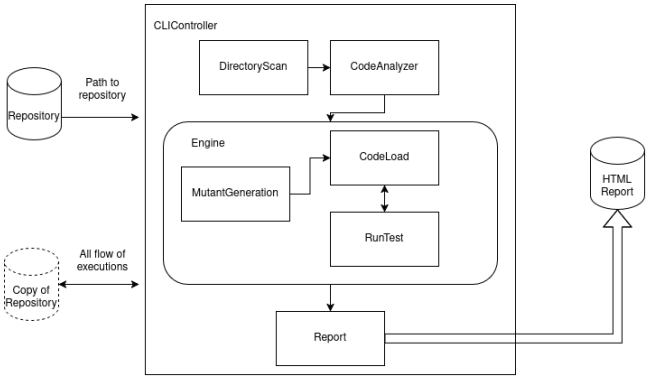

Figure 1: AutoMutSec architecture overview.

Once all mutation points are identified, the `Engine` component coordinates mutation generation and execution. The `MutantGeneration` component applies each mutation operator to the extracted expressions, producing the generated mutants.

Before executing any mutant, AutoMutSec validates that the original test suite passes without modifications. If any test fails at this stage, execution is interrupted and an error is reported, since a failing baseline would make mutation results uninterpretable. For each generated mutant, the `CodeLoad` component replaces the original authorization expression with its mutated version. The `RunTest` component then executes the test suite and collects the number of executed, failed, and skipped tests, as well as any execution errors. Results are extracted from JUnit report files, avoiding dependency on environment-specific log formats. After each execution, the original source code is restored before the next mutant is applied.

Finally, the `Report` component generates an HTML report listing all mutants, their status (killed or survived), and contextual information such as class name, method name, original expression, and mutated expression. The report also includes a summary with the overall mutation score and statistics.

5.2 Running Example

Consider the code snippet of Listing 17, where a controller contains an authorization expression.

Listing 17: Original authorization expression in DocumentController.

```
// DocumentController.java
@PreAuthorize("hasRole('ADMIN')")
@PostMapping("/document")
public void createDocument() { ... }
```

When AutoMutSec is executed with the repository path as input, it scans the project recursively, identifies `DocumentController.java`, and extracts the mutation point (class `DocumentController`, method `createDocument`, line 2, expression `hasRole('ADMIN')`). The mutation generation phase then produces the applicable mutants (Table 2); LSOR and APRO do not apply here, since they require a logical operator and a multi-argument function, respectively, neither of which is present.

Before applying mutations, AutoMutSec executes the original test suite to confirm that all tests pass. If the baseline is clean, the tool proceeds to the mutation execution phase. For each mutant, AutoMutSec replaces the expression at line 2 of `DocumentController.java`, executes the test suite, records the result, and restores the original file before moving to the next mutant. After all mutants are processed, AutoMutSec generates an informative HTML report with the full results, including the mutation score and the status of each individual mutant.

Table 2: Mutants generated for `hasRole('ADMIN')` by each operator.

Operator	Mutated Expression
ISIR	`hasRole('INVALID_ROLE')`
ARO	`hasRole('MANAGER')`
SETR	`hasAuthority('ADMIN')`
SITR	`hasRole('WRITE')`
LNSO	`!hasRole('ADMIN')`
PARO	`permitAll()`
DARO	`denyAll()`
EAAO	`hasRole(")`

6 Evaluation Study

This section presents the empirical evaluation of AutoMutSec. The goal of this study is to analyze the effectiveness and practical applicability of authorization-specific mutation testing for identifying weaknesses in test suites of Spring Security applications, from the perspective of developers and quality assurance teams, in the context of real-world Spring Boot projects.

The following research questions guide this evaluation:

- **RQ1: Can AutoMutSec identify weaknesses in the ability of test suites to detect authorization-related faults?** This question investigates whether the mutants generated by AutoMutSec expose deficiencies in existing test suites with respect to authorization fault detection. It is answered by analyzing the proportion of generated mutants that survive execution against the original test suites (mutation score).
- **RQ2: Do security-specific mutants reveal weaknesses that are not identified by traditional mutation operators?** This question evaluates whether AutoMutSec provides fault-detection capabilities beyond those of traditional mutation testing. It is answered by comparing the mutation scores obtained with AutoMutSec against those produced by PIT [5], a widely adopted mutation testing tool for Java. If security-specific mutants survive while PIT mutants are killed, this indicates that the test suite exercises general program behavior but insufficiently validates authorization-specific behavior.
- **RQ3: What is the execution cost of applying AutoMutSec?** This question investigates the practical feasibility of AutoMutSec in real-world projects. It is answered by measuring execution time and the number of generated mutants across the subject systems.

6.1 Subject Systems

Five Spring Boot projects were selected as subject systems. Four are open-source projects available on GitHub. The fifth is a private system used by a department of a university. Table 3 describes the selected systems. The systems were selected according to the following criteria: (i) the project uses Spring Boot and Spring Security; (ii) the source code contains at least one authorization expression that AutoMutSec can mutate; (iii) the test suite reaches and executes code elements protected by authorization expressions; and (iv) the project builds and executes successfully, with no failing test cases in the original test suite.

Table 3: Subject systems used in the evaluation.

Repository	KLOC	# of Tests	Intr. Cov.	Line Cov.
books[10]	7.793	169	87%	66%
gymstock[11]	5.223	106	59%	46%
rest-secure-spring-boot-starter[12]	3.097	78	97%	92%
saas-backend-starter[13]	4.288	45	29%	18%
Private repository	10.062	176	80%	68%

6.2 Experimental Setup

Experiments on the four open-source systems were conducted on a Google Cloud Platform e2-medium instance (2 vCPUs, 4 GB RAM, Debian GNU/Linux 12, amd64). The Private System, which depends on a containerized database, was executed on shared local hardware instead.

6.3 Results and Discussion

Table 4 summarizes the empirical results. In most of the evaluated systems, AutoMutSec mutation scores were low, indicating that existing test suites provide limited coverage of authorization-related behavior. Notably, no clear relationship was observed between traditional testing metrics, such as PIT mutation scores and test coverage, and the ability of test suites to detect authorization-specific faults. Several systems achieved considerably higher mutation scores with PIT than with AutoMutSec, suggesting that test suites may be effective at detecting conventional faults while remaining largely ineffective at validating authorization behavior.

6.3.1 RQ1: Can AutoMutSec identify weaknesses in test suites? The results show that AutoMutSec was able to expose authorization testing weaknesses across all evaluated systems. The most critical results were observed in `gymstock` and `saas-backend-starter`, where none of the generated mutants were detected by the existing test suites, yielding a mutation score of 0%.

A manual inspection of these projects revealed the main causes. In `gymstock`, most tests target the service layer, while authorization annotations are defined at the controller layer. As a result, the security infrastructure is never exercised during test execution. In `saas-backend-starter`, authorization annotations are present at the service layer, but the corresponding tests rely on mocks that bypass the protected methods entirely, preventing the authorization logic from being evaluated.

The `books` and `rest-secure-spring-boot-starter` systems achieved mutation scores of 49.25% and 58.82%, respectively, indicating that a substantial portion of authorization mutants still survived.

[10] https://github.com/aidanwhiteley/books
[11] https://github.com/asafeorneles/gymstock
[12] https://github.com/42BV/rest-secure-spring-boot-starter
[13] https://github.com/urosengineer/saas-backend-starter

Table 4: Comparison between AutoMutSec and PIT mutation testing results.

Repository	# of Mutants (AutoMutSec)	Mut. Score (AutoMutSec)	# of Mutants (PIT)	Mut. Score (PIT)	Exec. Time (PIT)	Exec. Time (AutoMutSec)	Slowdown
books	67	49.25%	1054	61%	2.35h	3.68h	1.5x
gymstock	1368	0%	399	37%	3min	24.28h	485x
rest-secure-spring-boot-starter	17	58.82%	163	80%	12min	1.63h	8.16x
saas-backend-starter	111	0%	327	19%	5min	5.85h	70.2x
Private System	182	82.42%	563	57%	1.61h	21h	19x

A concrete example from the books repository illustrates this. The class-level authorization annotation in BookSecureController defines the expression: hasAnyRole('ROLE_EDITOR','ROLE_ADMIN'). One generated mutant replaces the role-based check with an authority-based check: hasAnyAuthority('ROLE_EDITOR', 'ROLE_ADMIN'). This mutant survives because the test suite does not exercise a scenario where the semantic difference between roles and authorities affects the outcome. In contrast, a mutant that replaces a role name: hasAnyRole('NO_ROLE_EDITOR', 'ROLE_ADMIN') is killed because the modified rule prevents access in scenarios that existing tests expect to succeed.

Table 5 shows the distribution of generated and killed mutants per operator, offering a more granular view of these results. The pattern for PARO is the most clear: every PARO mutant survived in gymstock (38/38) and saas-backend-starter (11/11). No test in either system checks whether access restriction is enforced when an authorization rule is removed — arguably the most severe fault AutoMutSec can inject, since it implies any user could reach the protected resource undetected. books and the *Private System* perform differently, killing most PARO mutants (2/9 and 7/17), so at least some unauthorized-access checks exist there.

ARO alone accounts for 1,140 of gymstock's 1,368 mutants, none of which were killed, since ARO generates one mutant per valid role/authority substitution and gymstock defines authorization expressions with unusually large identifier sets. This concentration explains most of the system's mutant count. LSOR mutants appear only in saas-backend-starter (6, none killed) — compound expressions combining AND/OR are simply rare elsewhere in the systems. The *Private System* kills mutants consistently across operators (22/34 for ARO, 24/29 for APRO, 17/17 for LNSO, DARO, and EAAO), consistent with its 82.42% overall score (highest rate). Its test suite explicitly exercises authorization behavior under a real Spring Security context, successfully detecting the majority of injected authorization faults. This shows that AutoMutSec can confirm the adequacy of a well-tested authorization layer just as clearly as it exposes gaps in a poorly tested one.

A potential concern in mutation testing is the presence of equivalent mutants (mutants that alter the code without changing the observable behavior of the system). In the context of AutoMutSec, a representative example would be replacing hasRole('ADMIN') with hasAuthority('ROLE_ADMIN'): if the application assigns authorities using the ROLE_ prefix convention, this mutation may produce no behavioral difference, since Spring Security resolves hasRole(X) as ROLE_X internally. To assess this risk, we performed a sample validation of surviving mutants in our study and found no equivalent mutants among the cases inspected. Nevertheless, automatically identifying and excluding equivalent mutants remains a known open problem in mutation testing and a direction for future work in AutoMutSec.

To further validate the findings, we created additional tests for the books repository in the AutoMutSecTestsThatCanCapture MutantsTest class[14] These tests simulate different authorization scenarios using authenticated users with different roles and verify whether changes in security configuration produce unexpected access control behavior, based on the HTTP responses returned by the application. The additional tests successfully detected mutants that the original test suite had missed, confirming that the surviving mutants represent real gaps in authorization test coverage.

Our findings indicate that AutoMutSec can identify weaknesses in authorization test suites that would otherwise remain unnoticed. Furthermore, the absence of equivalent mutants in our sample inspection, together with the fact that manually written tests were able to kill previously surviving mutants, suggests that these weaknesses represent genuine testing gaps rather than artifacts of the mutation process. These results provide a positive answer to **RQ1**.

6.3.2 RQ2: Do security-specific mutants reveal weaknesses beyond traditional mutation testing? AutoMutSec and PIT target structurally distinct classes of faults. In the books repository, AutoMutSec exclusively mutates the contents of @PreAuthorize annotations, while PIT generates no mutations in authorization expressions, focusing instead on structural faults such as conditional statements, return values, and attribute initialization. The same pattern was observed across all evaluated repositories, indicating little to no overlap between the fault models of the two tools.

A concrete example from the UserController class in books illustrates this distinction (Listing 18). AutoMutSec generates mutants by modifying the authorization expression in @PreAuthorize by replacing the required role with another value, directly affecting the access-control policy. PIT generates no equivalent mutation for this annotation, instead, it targets structural elements in the method body. This confirms that the two tools are complementary rather than redundant.

Listing 18: Endpoint protected by a Spring Security authorization expression.

```
@DeleteMapping(value = "/users/{id}")
@PreAuthorize("hasRole('ROLE_ADMIN')")
public ResponseEntity<Object> deleteUserById( @PathVariable
      String id, Principal principal) {
... }
```

The quantitative results further support this conclusion. In systems with declarative authorization mechanisms, security-oriented mutants consistently exhibited higher survival rates than traditional mutants, suggesting that role validation and access-control enforcement are not adequately assessed by conventional mutation operators alone. Although the lack of overlap between AutoMutSec's and PIT's mutation targets is expected by design, the

[14] https://github.com/DovCaio/books/tree/automutsec-validation

Table 5: Per-operator mutant counts for each subject system.

	books		gymstock		rest-secure		saas		private repository	
Operator	Total	Killed	Total	Killed	Total	Killed	Total	Killed	Total	Killed
LSOR	0	0	0	0	0	0	6	0	0	0
LNSO	9	7	38	0	3	2	17	0	17	17
ISIR	11	7	38	0	2	1	11	0	34	29
ARO	7	0	1140	0	2	1	10	0	34	22
SETR	9	0	38	0	2	1	11	0	5	5
SITR	0	0	0	0	0	0	23	0	12	12
PARO	9	2	38	0	3	2	11	0	17	7
DARO	9	7	38	0	3	2	11	0	17	17
APRO	4	2	0	0	0	0	0	0	29	24
EAAO	9	8	38	0	2	1	11	0	17	17

relationship between their mutation scores is not. In four systems, AutoMutSec achieved lower scores than PIT, whereas in the *Private System* it achieved a substantially higher score (82.42% vs. 57%). This bidirectional pattern shows that authorization-specific test quality and general structural test quality are independent dimensions.

Therefore, neither mutation score reliably predicts the other. These findings provide a positive answer to **RQ2**: AutoMutSec exposes a complementary class of authorization-specific faults that lies outside the scope of PIT's mutation operators, supporting the combined use of both tools for a more comprehensive assessment of authorization test suites.

6.3.3 RQ3: What is the execution cost of AutoMutSec? AutoMutSec requires more execution time than PIT across all evaluated systems. This overhead is due to the tools execution model: each mutant requires a full, isolated test suite execution to ensure a consistent Spring Security context. PIT, in contrast, employs test selection and executes only the tests relevant to each mutant, reducing overall cost. The most extreme case was `gymstock`, where the high number of generated mutants (1,368) combined with the isolated execution model resulted in a slowdown of 485x relative to PIT.

The *Private System* (182 mutants, 19x slowdown) shows a markedly lower slowdown factor than `gymstock`, despite having a comparable absolute execution time. This is consistent with the system distinct testing infrastructure: unlike the other subject systems, which rely on the in-memory databases, *Private System*'s test suite runs against a real database with containerized dependencies, and tests were executed on shared local hardware rather than a dedicated cloud instance. These factors increase the cost of running the full test suite once, inflating both PIT's and AutoMutSec's absolute execution times proportionally. Since the slowdown factor compares AutoMutSec against PIT under the same infrastructure, it remains the more informative metric for assessing AutoMutSec's relative overhead, and 19x suggests that the isolated execution model is reasonably efficient when the mutant count is moderate. The `gymstock` case should therefore be understood as an outlier driven primarily by mutant count, rather than as representative of AutoMutSec's typical cost.

Despite this overhead, the isolated execution model performed by AutoMutSec provides reliable evaluation of authorization-specific behavior and avoids issues associated with dynamic in-memory modification of security-sensitive components. The approach is structurally parallelizable, since mutants are independent and could in principle be executed concurrently. However, we have not yet measured the resulting speedup, and quantifying it is a direction for future work.

We answer **RQ3** by observing that AutoMutSec incurs a higher computational cost than PIT. However, this additional cost is justified by its ability to evaluate authorization-specific faults and remains practical for periodic analyses in systems with a moderate mutant count, as observed in books, rest-secure-spring-boot-starter, and the Private System ($\leq$19x slowdown, $\leq$21h absolute time).

Overall, our findings carry two main implications. For developers, they highlight the importance of explicitly testing authorization requirements under a real Spring Security context, as general functional tests or reflection-based checks are insufficient to validate access-control behavior, even when line and instruction coverage are high. For mutation testing research, they reveal a structural blind spot in tools such as PIT: by focusing on language-level constructs, these tools leave declarative authorization policies outside their fault model, a gap that can be addressed through domain-specific operators such as those provided by AutoMutSec.

7 Threats to Validity

The five subject systems were selected based on explicit criteria, but may not be representative of the full diversity of Spring Security applications in terms of architectural styles, authorization complexity, and testing practices. In particular, systems were required to have test suites that already reach authorization-protected code, which may bias the results toward projects with better-than-average security testing practices. Replication with a larger and more diverse set of systems is needed to improve the generalizability of the findings.

The manual inspection of `gymstock` and `saas-backend-starter`, used to explain the 0% mutation scores, was performed by a the first author, introducing a risk of confirmation bias. Future replications should involve multiple reviewers.

The mutation score is used as a proxy for authorization test adequacy. Some surviving mutants may be equivalent, surviving not due to test suite weaknesses but because the mutation does not introduce a detectable behavioral difference. A sample validation found no equivalent mutants among the cases inspected. However, a systematic analysis of all surviving mutants was not performed.

The ten mutation operators were derived from the authors' experience and analysis of real-world bug reports and GitHub issues, rather than from a systematic fault taxonomy. Additional operators may be needed to represent authorization faults not captured by the current set.

Finally, PIT was selected as the baseline because it is the most widely adopted mutation testing tool for Java [5]. Other tools exist (e.g.,MuJava), but none target security-specific constructs. The comparison with PIT is therefore appropriate for establishing complementarity, though results may differ with other baseline tools.

8 Related Work

Mutation testing has been extensively studied as a technique for evaluating test suite effectiveness. The survey by Jia and Harman [9] highlights the predominance of operators targeting traditional programming faults (arithmetic expressions, conditional statements, logical operators), which do not address authorization-specific constructs. PIT [5], follows this same tradition: its operators target common implementation faults but do not cover authorization mechanisms expressed through Spring Security annotations.

The application of mutation testing to security-specific concerns has received growing attention. Shahriar and Zulkernine [13] proposed mutation operators targeting SQL injection vulnerabilities, demonstrating that domain-specific operators can expose security faults that conventional operators miss. More recently, Abdurrasyid et al. proposed 15 mutation operators for Broken Access Control vulnerabilities, specifically Insecure Path Lookup (IPL) and CSRF, derived through data flow analysis of PHP applications. Their results show that security-specific operators substantially improve test case quality for these vulnerability classes. AutoMutSec follows a similar motivation but targets a different context: authorization expressions in Spring Security Java applications, where access-control policies are defined declaratively through annotations rather than through imperative code constructs.

Security testing in a more general sense has been recognized as requiring approaches beyond conventional functional testing. Potter and McGraw [11] argue that security testing should be risk-driven, focusing on areas most prone to violations. Happe and Cito [7] similarly found authorization testing to be among the most time-consuming and least automated activities in web penetration testing. These observations reinforce the need for specialized techniques capable of systematically evaluating authorization behavior, a gap that AutoMutSec addresses through authorization-specific mutation operators for Spring Security.

9 Concluding Remarks

This work presented AutoMutSec, a mutation testing tool that introduces ten authorization-specific mutation operators targeting Spring Security configurations in Spring Boot applications. The empirical evaluation on five real-world projects generated 1,745 mutants, with scores ranging from 0% to 82.42%. AutoMutSec proved capable of both exposing authorization testing gaps and confirming the adequacy of well-tested authorization layers.

The comparison with PIT confirms that the two tools are complementary rather than redundant: AutoMutSec targets authorization expressions in annotations, a fault class entirely outside PIT's scope, while PIT continues to provide value for structural and language-level faults. Neither tool subsumes the other, and combining both offers a more complete picture of test suite adequacy than either alone.

Future work includes extending the proposed operators to other security frameworks and access-control models beyond Spring Security, reducing execution overhead through test selection and parallelization strategies, and conducting a systematic study of equivalent mutants to improve the precision of the mutation score. Integration with continuous integration pipelines, guided by the cost analysis in RQ3, is also a natural next step toward making authorization-oriented mutation testing part of standard quality assurance practice.

Artifact Availability

AutoMutSec is publicly available as an open-source tool. Its repository[15]. also includes the empirical data used in our study.

References

[1] Edrian S. Abduhari, Tadzher C. Shaik, Alsimar B. Adidul, Jimrashier H. Ladja, Ersin S. Saliddin, Akshay J. Adin, Fradzkhan A. Rumbahali, and Alnadzri B. Sali. 2024. Access Control Mechanisms and Their Role in Preventing Unauthorized Data Access: A Comparative Analysis of RBAC, MFA, and Strong Passwords. *Natural Sciences Engineering and Technology Journal* (2024). doi:10.37275/nasetjournal.v5i1.62

[2] Abdurrasyid, Yudistira Asnar, and Gusti Ayu Putri Saptawati. 2026. Mutation based improvement of security test case quality for broken access control. *Journal on Information Security* (2026).

[3] Richard Baker and Ibrahim Habli. 2012. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. (2012).

[4] Brasil. 2018. Lei Geral de Proteção de Dados Pessoais (Lei nº 13.709, de 14 de agosto de 2018). https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm. Accessed: June 8, 2026.

[5] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 449–452. doi:10.1145/2931037.2948707

[6] European Parliament and Council of the European Union. 2016. Regulation (EU) 2016/679 (General Data Protection Regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed: June 8, 2026.

[7] Andreas Happe and Jürgen Cito. 2023. Understanding hackers' work: An empirical study of offensive security practitioners. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1669–1680.

[8] Mazharul Islam, Sazzadur Rahaman, Na Meng, Behnaz Hassanshahi, Padmanabhan Krishnan, and Danfeng Daphne Yao. 2020. Coding Practices and Recommendations of Spring Security for Enterprise Applications. In *2020 IEEE Secure Development (SecDev)*. 49–57. doi:10.1109/SecDev45635.2020.00024

[9] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[10] M. Mythily, A. Samson Arun Raj, and Iwin Thanakumar Joseph. 2022. An Analysis of the Significance of Spring Boot in The Market. In *2022 International Conference on Inventive Computation Technologies (ICICT)*. IEEE. doi:10.1109/ICICT54344.2022.9850910

[11] B. Potter and G. McGraw. 2004. Software security testing. *IEEE Security Privacy* 2, 5 (2004), 81–85. doi:10.1109/MSP.2004.84

[12] Ana B. Sánchez, José A. Parejo, Sergio Segura, Amador Durán, and Mike Papadakis. 2024. Mutation Testing in Practice: Insights from Open-Source Software Developers. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3377378

[13] Hossain Shahriar and Mohammad Zulkernine. 2008. MUSIC: Mutation-based SQL injection vulnerability checking. In *2008 The Eighth International Conference on Quality Software*. IEEE, 77–86.

[14] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. 2007. Testing Security Policies: Going Beyond Functional Testing. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. 93–102. doi:10.1109/ISSRE.2007.27

[15] Li Zhong. 2023. A Survey of Prevent and Detect Access Control Vulnerabilities. *arXiv* abs/2304.10600 (2023). doi:10.48550/arXiv.2304.10600 Accessed: June 8, 2026.

[15] https://github.com/DovCaio/mutate_security_configs