Automated Detection of Broken Access Control Vulnerabilities: Methodologies, CWE Analysis, and Next-Generation Fuzzing

I. Foundational Taxonomy and Criticality of Access Control Failures

Broken Access Control (BAC) consistently represents the most critical security risk in modern application development, maintaining the top rank in the OWASP Top Ten list (A01:2021).[1] The high incidence rate, averaging 3.81% across tested applications and accounting for over 318,000 occurrences in contributed datasets, necessitates sophisticated, automated detection mechanisms.[1] Access control failures violate the principle of least privilege, allowing unauthorized information disclosure, modification, or destruction of data.[1] Failures manifest as the evasion of authentication or authorization checks, often by manipulating parameters, HTTP headers, or URLs (force browsing).[1]

A. Core Failure Types and CWE Hierarchy

To effectively develop and deploy automated detection tools, the spectrum of access control failures must be mapped onto the Common Weakness Enumeration (CWE) taxonomy. CWE-284, Improper Access Control, serves as the generalized parent class encompassing all failures to restrict access to resources.[3] However, practical vulnerability detection requires distinguishing between structural omissions and semantic logical flaws.

1. Structural and Semantic Authorization Flaws

The distinction between structural and semantic flaws drives the methodological requirements for automated testing.

- **CWE-862: Missing Authorization.** This represents a structural omission where a necessary authorization check is entirely absent from the code path intended to protect a resource. Automated static analysis (SAST) is moderately useful here for detecting commonly used authorization idioms and analyzing configuration files (such as Apache .htaccess directives). However, SAST tools generally face significant difficulty when authorization schemes are custom-implemented or dynamically determined, often leading to false positives when encountering functionality intentionally designed to be accessible to all users.[4]
- **CWE-863: Incorrect Authorization.** This represents a deeper, semantic flaw. An authorization check is present, but the underlying logic is faulty, resulting in the wrong permissions being granted or the check being insufficient to enforce the intended policy.[5] Detecting CWE-863 requires moving beyond simple code presence checks and necessitates dynamic analysis or policy verification capable of understanding application logic and context.

2. Specialized Access Control Failures

Two other highly relevant CWEs represent common and high-impact exploitation vectors, particularly in API environments:

- **CWE-639: Authorization Bypass Through User-Controlled Key (IDOR/BOLA).** This is often termed Insecure Direct Object Reference (IDOR) or Broken Object-Level Authorization (BOLA), and is a primary focus for modern API security.[6] CWE-639 involves the retrieval of a user record or resource based on a key value that is under user control (e.g., in a URL parameter or hidden field). The authorization process fails to ensure the authenticated user performing the operation has the requisite entitlements to access that specific resource. This is a common manifestation of "horizontal authorization," where two users possess the same privilege level but must be prevented from accessing each other's data.[6]
- **CWE-732: Incorrect Permission Assignment for Critical Resource.** This focuses on misconfiguration at the operating system or application environment level, where improper permissions are assigned to a critical resource (e.g., sensitive files or

configurations).[8] Detection of this flaw often requires analyzing static configuration files rather than dynamic code execution.

CWE ID	CWE Name	BAC Specificity/Relationship	Primary Detection Paradigm
CWE-284	Improper Access Control	The generalized root class for all failures to restrict access.	Policy Modeling & Auditing
CWE-862	Missing Authorization	Structural omission: A required check is entirely absent.	SAST/ML (SRM Classification)
CWE-863	Incorrect Authorization	Semantic flaw: Authorization logic is present but incorrectly implemented.	Hybrid/Grey-Box Fuzzing (Logic Check)
CWE-732	Incorrect Permission Assignment	Misconfiguration of permissions for underlying system resources (files, etc.).	Static Configuration Analysis
CWE-639	Authorization Bypass (IDOR/BOLA)	Horizontal or Vertical privilege escalation via object key manipulation.	DAST/LLM-Guided Fuzzing (Reference Mutation)

B. The Nexus of API Security and Chained Exploitation

The proliferation of APIs has centered the focus of BAC detection on CWE-639 (BOLA). Automated tools must specifically target object key manipulation in endpoints, moving beyond generic checks for authentication status. This requirement stems from the fact that modern applications, especially those relying on microservices, decentralize access control, placing greater reliance on checking direct object identifiers.[6]

Furthermore, BAC vulnerabilities rarely exist in isolation; they often serve as critical stepping stones for exploiting other, more severe flaws. An access control flaw that grants an attacker higher-level access (privilege escalation) provides the necessary context to launch secondary, high-impact exploits, such as command injection.[2] Therefore, automated systems must not only identify BAC but also evaluate its potential to facilitate exploit chaining.

II. Automated Detection Paradigms: From Static Analysis to ML-Policy Verification

Automated BAC detection has evolved significantly, moving from heuristic-based black-box testing towards sophisticated techniques that integrate machine learning and deep program analysis to address the inherent complexity of authorization logic.

A. Static Analysis and Machine Learning for Security-Relevant Methods

Traditional Static Application Security Testing (SAST) is effective for low-hanging fruit related to CWE-862, successfully identifying widely adopted authorization libraries and configuration idioms.[4] However, the inability of standard SAST to comprehend complex application flow or business logic limits its effectiveness against semantic flaws (CWE-863).

To inject semantic awareness into static analysis, advanced tools like SWANASSIST integrate machine learning (ML) to classify Security-Relevant Methods (SRM).[10] This approach utilizes 25 types of binary ML features, instantiated into 206 concrete features, based on intra-procedural data flows, method signatures, parameter types, and modifiers.[10]

The classification process is executed in two iterations: first, all methods in the analyzed program are classified into general SRM types (sources, sinks, sanitizers, or authentication methods); second, the identified SRMs are refined into specific CWEs, including the structurally focused CWE-862 and the semantically focused CWE-863.[10] For instance, a feature like

methodClassContainsOAuth helps indicate a method's intent related to authentication or authorization.[10] By recognizing the logical

intent of a method through these features, this approach aims to detect instances where the intent exists but the implementation is incorrect (CWE-863), scaling SAST beyond simple pattern matching.

B. Dynamic Analysis and API Specification Processing

Dynamic Application Security Testing (DAST) tools are essential for testing the application at runtime. Commercial platforms like Burp Suite Professional, along with open-source alternatives like OWASP ZAP, provide invaluable resources for penetration testing.[11] Specialized DAST extensions, such as Burp Autorize and Auth Analyzer, further aid in managing testing across multiple user roles and customizing request parameters (headers, cookies) to probe for bypass issues.[12]

For addressing Insecure Direct Object Reference (IDOR) and Broken Object-Level Authorization (BOLA), a critical subset of CWE-639, a specialized approach involves the processing of API specifications.[7] Given that BOLA vulnerabilities are inherently tied to the predictable, structured nature of RESTful APIs, automation can be achieved by analyzing system specifications (e.g., OpenAPI). This methodology involves:

1. Systematizing IDOR/BOLA attack patterns derived from literature review and real-world cases.
2. Grouping these patterns based on common attributes (endpoint properties, attack techniques).
3. Mapping these group features directly to the properties defined within the OpenAPI specification.[7]

This method generates targeted attack vectors based on documented API structure, effectively transforming generic DAST into optimized BOLA detection tailored to specific endpoints and object parameters.[7]

C. Hybrid Detection Architectures

Purely static analysis yields too many false positives, while traditional black-box dynamic analysis struggles with coverage and business logic comprehension. A hybrid approach, combining the depth of white-box analysis with the validation of black-box testing, is often required to achieve a superior detection balance.[13]

A robust hybrid scanner typically uses a black-box technique to improve detection precision and minimize the false alarm rate.[13] The fundamental components include a

Crawler, which establishes a connection, browses forms, and gathers evidence about the application's structure, and a **Fuzzing component**, which handles responses and identifies exposures.[13] This synergy is crucial for identifying access control checks related to Security Sensitive Operations (SSOs), such as database operations, across complex applications, particularly those based on frameworks like PHP.[14]

D. Machine Learning for Access Control Policy Verification

Addressing access control vulnerabilities requires proactive measures taken before implementation. The National Institute of Standards and Technology (NIST) has proposed an innovative technique for access control policy verification using a Machine Learning Classification algorithm, specifically Random Forest Classification (RFC).[15] This method is designed to overcome the difficulties associated with traditional policy verification techniques, which struggle with comprehensive test coverage, reliable oracles, and system translation.[15]

1. The RFC Policy Verification Mechanism

This ML approach enables detection of inconsistencies and semantic faults directly within the policy rules themselves. The policy rules are treated as the training data, utilizing the attributes (subject attributes, actions, object attributes) as feature values. The target for classification is the access permission assigned to the rule (e.g., grant or deny).[15]

The RFC algorithm generates ensembles of decision subtrees representing multiple model policy rules. Inconsistencies—where the model's prediction does not match the training data—indicate flaws or conflicts within the policy rules.[15] This policy verification approach represents a significant step forward, moving detection upstream to find flaws in the *design* (CWE-863 root cause) rather than just the *implementation*.[15]

2. Data Preparation for Policy Verification

Successful application of the RFC algorithm demands meticulous data preparation. Policy rules must be cleaned, ensuring only enforceable rules are present. Crucially, any rule containing OR relations for actions or object attribute values must be broken down and dispersed into separate sub-rules, each occupying a distinct row in the data table.[15] This structural requirement ensures that the RFC algorithm accurately models the policy logic, as its binary tree structure evaluates attributes and actions based on Boolean logic paths. In contrast, AND relations can coexist within the same row, corresponding to true paths in a single tree branch.[15] By checking the rule logic directly, the RFC method is both efficient and capable of identifying rule conflicts that developers might overlook in specification documents.

III. The Cutting Edge: LLM-Guided Grey-Box Fuzzing for BAC

Despite advancements in SAST, DAST, and policy verification, automated detection of access control flaws is hindered by two persistent challenges that prevent tools from effectively finding semantic flaws like CWE-863 and CWE-639.

A. Persistent Challenges in Automated BAC Detection

The challenges force state-of-the-art detection methodologies to adopt grey-box techniques that leverage both code knowledge and runtime interaction.

- **The Test Oracle Problem:** Unlike injection vulnerabilities, which often result in explicit errors or crashes, BAC vulnerabilities frequently result in "silent flaws".[16] An unauthorized request may successfully access or modify data, yet the server returns a benign status code (e.g., 200 OK), masking the breach.[14] Determining if an unauthorized action was truly denied or if the denial simply stemmed from an incorrect input format is a critical ambiguity in traditional black-box testing.

- **Generating Semantically Valid Inputs:** Automated testing requires generating test cases that involve a series of user interactions and semantic understanding of the business logic.[14] Random fuzzing of inputs often results in the application rejecting the request with HTTP 4xx errors, preventing the malicious input from ever reaching the authorization check logic deep within the application.[16] Exploiting BOLA (CWE-639) requires manipulating parameters with semantic meaning (like object IDs), which generic mutation struggles to achieve reliably.[17]

B. BACFuzz: A Solution using LLM Guidance and SQL Oracles

Recent research introducing BACFuzz, a grey-box fuzzing framework for PHP applications, provides a breakthrough by addressing both the semantic input generation and the oracle problem for BOLA and Broken Function-Level Authorization (BFLA).[16]

1. LLM-Guided Parameter Analysis

BACFuzz leverages Large Language Models (LLMs) to transform the mutation process from a brute-force effort into targeted, intelligent testing.[17] The LLMs act as semantic navigators, analyzing collected HTTP request traffic to identify

semantically important parameters—those references (e.g., user IDs, object handles) that are likely to reveal privilege differences between users.[17]

This LLM guidance significantly reduces the mutation search space by focusing fuzzing efforts solely on parameters referencing protected objects or functionalities. This results in the generation of semantically valid test cases that frequently bypass front-end 4xx response rejections and successfully reach the authorization logic, a crucial step for exploiting CWE-639.[17]

2. SQL-Based Oracle Checking

The most profound innovation in the BACFuzz methodology is its solution to the authorization oracle problem. The framework employs a grey-box approach using **lightweight code**

instrumentation (function hooking) to monitor original PHP functions, particularly those related to backend SQL queries.[17]

The core verification mechanism inspects the backend SQL queries issued by the application.[16] The crucial confirmation logic is established by determining if the mutated, unauthorized input value successfully appears in the constructed SQL query. If the unauthorized input flows into the protected operation's database query, it confirms that the authorization validation was bypassed, definitively identifying the silent flaw regardless of the benign HTTP response code.[16] This provides the necessary ground truth confirmation that traditional dynamic testing lacks.

3. Performance and Related Techniques

BACFuzz has demonstrated high efficacy, successfully detecting 16 out of 17 known BAC issues across 20 real-world web applications (including 15 CVE cases) and uncovering 26 previously unknown BAC vulnerabilities, all with reported low false positive rates.[16] This performance demonstrates that successful automated detection for advanced semantic flaws mandates this synthesis of LLM-driven intelligence, grey-box instrumentation, and dynamic runtime monitoring via a reliable oracle.

Similarly, LLMs are being applied in specialized contexts, such as ACBreaker, which automatically detects BAC in IoT protected interfaces by analyzing attack vectors like HTTP path, parameter, and header manipulation.[2]

Table 2: State-of-the-Art Automated BAC Detection Approaches

Approach Category	**Example Technology/Tool**	**Primary Technique**	**Targeted BAC Subtypes**	**Key Innovation/Mechanism**
Static/ML Hybrid	SWANASSIST	Machine Learning (SRM Classification)	CWE-862, CWE-863	Uses 206 features to inject semantic awareness into static analysis for identifying security intent.[10]

DAST/Specification	OpenAPI Processor	Pattern Matching / Specification Analysis	IDOR/BOLA (CWE-639)	Maps real-world attack patterns to API specification properties for targeted endpoint testing.[7]
Policy Verification	NIST RFC Approach	Machine Learning (Policy Consistency)	Policy Flaws (Pre-Deployment)	Uses Random Forest to check the logic of policy rules directly for conflicts and semantic errors.[15]
Grey-Box Fuzzing	BACFuzz	LLM-Guided Fuzzing + SQL Oracle	BOLA, BFLA (CWE-639, CWE-863)	Overcomes the oracle problem by monitoring backend database queries and uses LLMs for semantic input generation.[16]

IV. Case Studies: Mapping CVEs to Automated Detection Requirements

Analysis of recent, high-profile Common Vulnerabilities and Exposures (CVEs) illustrates how automated techniques must address specific attack patterns associated with BAC. Real-world exploitation confirms that the CWE taxonomy accurately models operational security defects.

A. Semantic vs. Structural Flaw Examples

The operational differences between CWE-862 and CWE-863 are clearly demonstrated in practical exploitation scenarios:

- **Missing Authorization (CWE-862 / BFLA): CVE-2021-39931**, affecting Gitlab's branch deletion endpoint, serves as a textbook example of Broken Function-Level Authorization (BFLA).[5] The endpoint lacked any role or permission check, enabling any user with a valid token—regardless of their security scope—to successfully execute the DELETE request.[5] Automated detection systems focusing on path coverage (CWE-862) or BFLA (testing high-privilege functions with low-privilege tokens) are designed to find such omissions.
- **Incorrect Authorization (CWE-863 / Privilege Escalation): CVE-2023-3290**, a vulnerability found in Mealie v2.2.0, demonstrates a failure in logical policy enforcement (CWE-863).[5] Group managers were permitted to edit their
own permissions via the /api/households/permissions endpoint. By submitting a payload to modify their user entry and add administrative rights, they could effectively self-escalate their privileges.[5] Automated tools aiming to find this flaw must incorporate deep semantic logic checks to identify situations where user-controlled keys are tied to security-sensitive operations, ensuring the application explicitly prohibits self-escalation or modifications outside of defined role boundaries.

B. Chained Exploitation and Critical Bypass

The severity of a BAC flaw is often amplified when it facilitates unauthenticated access or acts as the precursor to remote code execution (RCE).

- **Authentication Bypass (CVE-2023-46805):** This high-severity flaw in Ivanti Secure and Policy Secure gateways allows a remote attacker to access restricted resources by bypassing control checks.[9] This vulnerability, which is fundamentally a high-impact BAC failure (likely CWE-862 or CWE-863), removes the authentication boundary.
- **Facilitating RCE:** The true criticality of CVE-2023-46805 was realized when it was chained with CVE-2024-21887 (a command injection vulnerability).[9] The resulting attack chain allowed exploitation that did
not require authentication, enabling the threat actor to craft malicious requests and execute arbitrary commands on the system.[9] This case highlights that automated detection must prioritize BAC flaws that eliminate fundamental security boundaries, as such vulnerabilities exponentially increase the risk posed by subsequent flaws. Monitoring runtime security and detecting these multi-stage attack patterns is critical for

defense.

C. Configuration and Permission Assignment (CWE-732)

Vulnerabilities tied to system configurations, such as improper file or database permissions, represent a unique challenge that traditional dynamic analysis struggles to address.

- **Incorrect Permission Assignment (CWE-732 Context):** An exemplary case is CVE-2023-21409, affecting the AXIS License Plate Verifier, where insufficient file permissions allowed unprivileged users to gain unauthorized access to unencrypted administrator credentials.[20] This class of flaw cannot typically be discovered by generating malicious HTTP requests. Instead, it requires sophisticated SAST or configuration analysis tools capable of parsing server-side directives and operating system settings to ensure critical resources are only accessible by appropriately privileged entities.[4] While specific exploitation details for CVE-2023-3286 and CVE-2024-55070 require further specialized analysis, the contextual requirement for automated tools remains centered on configuration integrity for CWE-732.

V. Strategic Implementation, Limitations, and Future Directions

The contemporary understanding of automated BAC detection necessitates a methodological convergence, utilizing specialized tools at different stages of the Software Development Life Cycle (SDLC) to cover the full range of CWEs.

A. Strategic Integration of Advanced Methodologies

Security architects must adopt a proactive, layered defense strategy:

1. **Shift Left with Policy Verification:** The integration of ML-based policy verification (e.g., the NIST RFC approach) into the design and early development phases ensures logical consistency and helps mitigate root policy conflicts (CWE-863) before code implementation.[15] This approach ensures the policy specification is sound before

resource investment in coding begins.

2. **Continuous Grey-Box Fuzzing:** The deployment pipeline must incorporate advanced grey-box fuzzing, such as the BACFuzz methodology, to continuously test deployed APIs for BOLA (CWE-639) and BFLA (CWE-863).[16] This methodology, leveraging the SQL-based oracle, provides the necessary accuracy and low false positive rates required for production environments.[17]
3. **Layered DAST and Auditing:** While specialized tools address complexity, general DAST tools like Burp and ZAP remain valuable. Organizations should adopt a combined vulnerability scanning approach (the "Union List" framework) to aggregate and correlate results from multiple DAST scanners, which has been shown to exhibit greater accuracy and recall compared to reliance on individual scanners.[21] Crucially, regular, systematic access control audits must be conducted to ensure user permissions strictly adhere to the principle of least privilege (Role-Based Access Control).[22]

B. Persistent Challenges in Automation

Despite technological progress, several persistent challenges limit the scope and efficiency of automated BAC detection:

- **Business Logic Dependence:** Automated testing struggles immensely when authorization logic is highly dependent on complex, custom business rules that are unique to the application.[14] Defining an accurate test oracle and generating inputs that fulfill the necessary multi-step business requirements remains resource-intensive.
- **Zero-Day Detection:** Automated systems primarily detect deviations from static policies or known attack patterns. The challenge of detecting entirely new web application outbreaks and zero-day BAC vulnerabilities requires continuous research into algorithm improvement.[13]
- **Governance and Policy Adherence:** Technical solutions are insufficient if they are not integrated into a strong governance framework. Effective mitigation requires adherence to core principles, such as the CIA Triad (Confidentiality, Integrity, Availability), ensuring these principles are woven into the cybersecurity strategy.[22]

C. Future Trajectories

Future automation efforts are increasingly focusing on utilizing Large Language Models (LLMs) to bridge the semantic gap in security testing. While LLM agents pose a potential threat by autonomously conducting cyberattacks [23], they also represent a crucial defensive capability.

LLMs are being explored for policy synthesis, attempting to infer and formalize access control policies from natural language descriptions contained in project documents.[24] While early results indicate that synthesized policies can be syntactically valid, they may differ significantly in permissiveness compared to the actual ground truth policy, indicating ongoing challenges in policy inference accuracy.[24] However, the role of LLMs as semantic navigators—guiding grey-box fuzzers toward meaningful exploitation vectors (CWE-639)—is already established as a major advance in detection efficacy.[17]

VI. Conclusions and Recommendations

The transition of Broken Access Control to the top of the criticality list demands sophisticated, converged detection strategies. The analysis confirms that no single automated methodology is sufficient to reliably detect all classes of BAC flaws (CWE-862, CWE-863, CWE-639, CWE-732).

Structural flaws (CWE-862) are best addressed by advanced, feature-rich SAST approaches integrated with ML-based SRM classification (like SWANASSIST). However, semantic flaws (CWE-863, CWE-639)—the majority of contemporary API risks—mandate the use of dynamic techniques that can interpret and verify complex logic. The development of LLM-guided grey-box fuzzing frameworks, utilizing SQL-based oracles to monitor unauthorized database interaction, is essential for reliably detecting the silent flaws inherent in authorization testing. This convergence of LLM-driven intelligence, grey-box instrumentation, and objective runtime confirmation represents the state-of-the-art required to manage the high prevalence and impact of BAC vulnerabilities, especially those facilitating complex attack chains like the Ivanti RCE exploit.

Organizations must prioritize investment in policy verification tools that validate the authorization *design* (NIST RFC approach) and deploy continuous API security platforms capable of runtime BOLA detection, while maintaining robust, regularly audited access control policies aligned with the principle of least privilege.[22]

Works cited

1. A01 Broken Access Control - OWASP Top 10:2021, accessed October 4, 2025, https://owasp.org/Top10/A01_2021-Broken_Access_Control/
2. Large Language Model-Powered Protected Interface Evasion: Automated Discovery of Broken Access Control Vulnerabilities in Internet of Things Devices -

PMC - PubMed Central, accessed October 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC12074161/
3. CWE-284: Improper Access Control (4.18) - Common Weakness Enumeration - MITRE Corporation, accessed October 4, 2025, https://cwe.mitre.org/data/definitions/284.html
4. CWE-862: Missing Authorization - Common Weakness Enumeration - MITRE Corporation, accessed October 4, 2025, https://cwe.mitre.org/data/definitions/862.html
5. BLA9:2025 - Broken Access Control (BAC) - OWASP Foundation, accessed October 4, 2025, https://owasp.org/www-project-top-10-for-business-logic-abuse/docs/the-top-10/broken-access-control
6. CWE-639: Authorization Bypass Through User-Controlled Key, accessed October 4, 2025, https://cwe.mitre.org/data/definitions/639.html
7. [2201.10833] Automatic detection of access control vulnerabilities via API specification processing - arXiv, accessed October 4, 2025, https://arxiv.org/abs/2201.10833
8. CWE-732: Incorrect Permission Assignment for Critical Resource - MITRE Corporation, accessed October 4, 2025, https://cwe.mitre.org/data/definitions/732.html
9. CVE-2023-46805 (Authentication Bypass) & CVE-2024-21887 (Command Injection) for Ivanti Connect Secure and Ivanti Policy Secure Gateways, accessed October 4, 2025, https://forums.ivanti.com/s/article/CVE-2023-46805-Authentication-Bypass-CVE-2024-21887-Command-Injection-for-Ivanti-Connect-Secure-and-Ivanti-Policy-Secure-Gateways
10. SWANASSIST: Semi-Automated Detection of Code ... - Eric Bodden, accessed October 4, 2025, https://www.bodden.de/pubs/ase19swanAssist.pdf
11. Best Tools to Identify Broken Access Control in APIs | Prophaze Blog, accessed October 4, 2025, https://prophaze.com/blog/broken-access-control-api-tools/
12. Identifying Broken Access Controls: A Comprehensive Guide - Tevora, accessed October 4, 2025, https://www.tevora.com/threat-blog/finding-broken-access-controls/
13. A HYBRID APPROACH TO DETECT SECURITY VULNERABILITIES IN WEB APPLICATIONS, accessed October 4, 2025, https://ijcsmc.com/docs/papers/February2022/V11I2202212.pdf
14. A Survey of Prevent and Detect Access Control Vulnerabilities - arXiv, accessed October 4, 2025, https://arxiv.org/pdf/2304.10600
15. Machine Learning for Access Control Policy Verification, accessed October 4, 2025, https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8360.pdf
16. BACFuzz: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications - arXiv, accessed October 4, 2025, https://arxiv.org/html/2507.15984v1
17. (PDF) BACFuzz: Exposing the Silence on Broken Access Control ..., accessed October 4, 2025,

https://www.researchgate.net/publication/393923712_BACFuzz_Exposing_the_Silence_on_Broken_Access_Control_Vulnerabilities_in_Web_Applications
18. [2507.15984] BACFuzz: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications - arXiv, accessed October 4, 2025, https://www.arxiv.org/abs/2507.15984
19. The Most Exploited Vulnerabilities of 2024 - Arctic Wolf, accessed October 4, 2025, https://arcticwolf.com/the-most-exploited-vulnerabilities-of-the-year/
20. Security Advisories - Axis Documentation, accessed October 4, 2025, https://help.axis.com/en-us/security-advisories
21. Enhancing Web Application Security through Automated Penetration Testing with Multiple Vulnerability Scanners - MDPI, accessed October 4, 2025, https://www.mdpi.com/2073-431X/12/11/235
22. The Risks of Broken Access Control Explained: Vulnerabilities, Examples & Best Practices, accessed October 4, 2025, https://www.activestate.com/blog/the-risks-of-broken-access-control-explained-vulnerabilities-examples-best-practices/
23. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities - arXiv, accessed October 4, 2025, https://arxiv.org/pdf/2503.17332?
24. Synthesizing Access Control Policies using Large Language Models - arXiv, accessed October 4, 2025, https://arxiv.org/html/2503.11573v1
25. Statement Recognition of Access Control Policies in IoT Networks - MDPI, accessed October 4, 2025, https://www.mdpi.com/1424-8220/23/18/7935