BACFuzz: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications

I Putu Arya Dharmaadi
University of Groningen
Groningen, Netherlands
arya.dharmaadi@rug.nl

Mohannad Alhanahnah
Chalmers University
Gothenburg, Sweden
mohannad.alhanahnah@chalmers.se

Van-Thuan Pham
The University of Melbourne
Melbourne, Australia
thuan.pham@unimelb.edu.au

Fadi Mohsen
University of Groningen
Groningen, Netherlands
f.f.m.mohsen@rug.nl

Fatih Turkmen
University of Groningen
Groningen, Netherlands
f.turkmen@rug.nl

Abstract

Broken Access Control (BAC) remains one of the most critical and widespread vulnerabilities in web applications, allowing attackers to access unauthorized resources or perform privileged actions. Despite its severity, BAC is underexplored in automated testing due to key challenges: the lack of reliable oracles and the difficulty of generating semantically valid attack requests. We introduce **BACFuzz**, the first gray-box fuzzing framework specifically designed to uncover BAC vulnerabilities, including Broken Object-Level Authorization (BOLA) and Broken Function-Level Authorization (BFLA) in PHP-based web applications. **BACFuzz** combines LLM-guided parameter selection with runtime feedback and SQL-based oracle checking to detect silent authorization flaws. It employs lightweight instrumentation to capture runtime information that guides test generation, and analyzes backend SQL queries to verify whether unauthorized inputs flow into protected operations. Evaluated on 20 real-world web applications, including 15 CVE cases and 2 known benchmarks, **BACFuzz** detects 16 of 17 known issues and uncovers 26 previously unknown BAC vulnerabilities with low false positive rates. All identified issues have been responsibly disclosed, and artifacts will be publicly released.

CCS Concepts

• **Security and privacy → Access control**; • **Software and its engineering → Software testing and debugging**.

Keywords

Broken Access Control, Web, Grey-box Fuzzing, LLM, SQL, PHP

ACM Reference Format:
I Putu Arya Dharmaadi, Mohannad Alhanahnah, Van-Thuan Pham, Fadi Mohsen, and Fatih Turkmen. 2025. **BACFuzz**: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications. In *Proceedings of XXX (Under peer-review)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The widespread adoption of web applications has brought associated security risks to the forefront, drawing significant attention from both academia and industry. To address these risks, OWASP has published two influential guides: the OWASP Top 10 Web Application Security Risks [29] and the OWASP API Security Top 10 [28]. Last updated in 2023, both documents identify Broken Access Control (BAC) as the most prevalent and critical security flaw.

BAC enables attackers to access unauthorized functionality or data—often resulting in vertical privilege escalation [35]. For example, a regular user might overwrite an administrator's email address or invoke privileged actions such as deleting user accounts. Several recent high-impact incidents involving BAC vulnerabilities (see Section 2.2) underscore the urgency of robust access control testing.

While OWASP provides guidelines for testing access control [30], automated tools capable of doing so with minimal human effort remain lacking. Manual analysis is labor-intensive and difficult to scale for complex applications [19], and most existing fuzzers primarily target crash-inducing bugs [3, 50]. Recent surveys [9, 11, 48] confirm that fuzzing research rarely addresses logical vulnerabilities such as BAC.

Our analysis of BAC characteristics (see Section 3) identifies two main challenges that hinder fuzzing for BAC. First, designing a test oracle—i.e., determining whether a request triggers a vulnerability—is difficult because BAC violations typically do not cause crashes or produce explicit error messages. Detecting such silent flaws often requires deep domain knowledge, as seen in recent research on SQL injection [44], excessive data exposure [33], and DNS resolver bugs [49]. Heuristics based on HTTP response codes or messages are often unreliable due to ambiguous or generic server feedback. Second, API requests often include many parameters with diverse data types and dependencies, resulting in a vast search space. Generating inputs that are both valid and semantically meaningful is non-trivial, as malformed requests are usually rejected.

To address the oracle challenge, we draw inspiration from prior work on injection flaws [44]. Once a request is accepted, the database-backed web application typically parses it, performs validation and authorization checks, and then constructs backend SQL queries. If a mutated input value appears in a resulting query, it suggests that some validation—potentially an authorization check—was bypassed. While this is a necessary condition for BAC, sufficiency depends on the context; for example, in Broken Function Level Authorization (BFLA) cases, the operation may not even be visible in the current user's interface. Based on this insight, we define a set of rules to reliably detect BAC violations (see Section 4.3).

To address the input generation challenge, we focus on the two most common BAC types: Broken Object Level Authorization (BOLA) and Broken Function Level Authorization (BFLA). Rather than mutating all parameters equally, we target those referencing protected objects or functionalities—i.e., those likely to reveal privilege differences between users. We collect traffic from users with different roles, label the requests, and leverage Large Language Models (LLMs) to identify semantically important parameters. LLMs are well-suited to this task due to their reasoning capabilities and ability to understand structured data. This significantly reduces the mutation space while preserving request validity.

Building on these ideas, we present **BACFuzz**, the first automated fuzzing tool specifically designed to detect BAC vulnerabilities in PHP-based web applications. **BACFuzz** uses a grey-box fuzzing approach, leveraging runtime feedback collected via lightweight code instrumentation—a method widely regarded as effective and scalable in modern fuzzing [20]. Our instrumentation uses function hooking techniques to monitor original PHP functions related to SQL queries, enabling precise feedback and oracle validation.

We evaluate **BACFuzz** on 20 PHP-based Web Under Test (WUT) applications. It successfully reproduces 16 of 17 known BAC issues and uncovers 26 previously unknown vulnerabilities, all of which have been responsibly disclosed. While instrumentation introduces minimal overhead, it enables the generation of semantically valid, high-coverage test cases—frequently bypassing 4xx response rejections. Our results demonstrate that **BACFuzz** is effective, scalable, and practical for uncovering BAC vulnerabilities in real-world applications.

In summary, this paper makes the following contributions:

1. **Novel Approach**: We propose the first grey-box fuzzing method specifically designed to automatically detect object- and function-level BAC vulnerabilities in web applications.
2. **New Exploitation Strategy**: We introduce an active checker module that applies advanced grey-box fuzzing techniques, including LLM-guided parameter analysis and reference mutation, to exploit captured HTTP requests.
3. **Novel Oracle Verification**: We develop a verification module that inspects SQL queries issued by the application. If mutated inputs appear in these queries and vulnerability type-specific conditions are met, we confirm that the target object or function is vulnerable.
4. **New BAC Vulnerability Dataset**: We curate and release a dataset of web applications with known CVE-reported BAC vulnerabilities, addressing the lack of existing benchmarks for BAC detection tools.

```
POST /wp-admin/admin-ajax.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

action=wcfm_ajax_controller&controller=wcfm-customers-
manage&.............................................customer_id=1.......................................
```

Figure 1: An HTTP Request exploiting CVE-2024-8290. Normally, the script on the web client sets the *customer_id* parameter to have a default value of 0, meaning the creation of a new customer. Arbitrarily changing the value to 1 leads the submitted data to replace the original data of a user with the ID of 1.

2 Background and Motivation

2.1 Background

According to the definition provided by OWASP [29], BAC refers to a vulnerability that occurs when a web application allows a certain access request to read or modify resources that it should not be able to. One specific example of BAC is **IDOR (Insecure Direct Object References)**, which enables malicious parties to access protected resources by sending resource identifiers through user-controlled parameters. There are two risks related to IDOR, as follows.

BOLA—Broken Object Level Authorization (API1:2023). BOLA occurs as inadequate access controls allow a user to manipulate or access data objects that they are not permitted to view or modify. For example, assume that a web portal application stores *products* from different sellers and only the corresponding seller is allowed to modify them. When a seller can modify a product that belongs to another seller by sending the ID reference of the product, the web application is said to have BOLA. Another example is explained in Figure 1.

BFLA—Broken Function Level Authorization (API5:2023). BFLA occurs when a web application fails to enforce proper authorization checks on user access to specific web functions (e.g., API endpoint). For example, consider the case in which a *button to create a new object* on a web page is removed when non-admin users access the page since only administrators are allowed to create a new object. The application is said to have BFLA if the web server accepts/executes a direct HTTP request originating from non-admin users to create an object.

BOLA vs BFLA. When certain HTTP requests are supposed to be available only for some users, and the server executes the same requests even if they originate from other users, it is **BFLA**. On the other hand, **BOLA** arises when the request is indeed available for a user yet the user modifies a parameter identifying an object with another ID reference that is not available his/her pages, and the server executes the request.

2.2 Motivating Examples

This section presents two real-world examples of BAC-related vulnerabilities to motivate the relevance of automated BAC vulnerability detection.

CVE-2024-8290 [7], disclosed in September 2024, represents a critical BAC vulnerability within the *wc-frontend-manager* plugin for WordPress. This CVE report has a significant impact on the score (CVSS 8.8 with high severity), affecting approximately 20,000 users, according to the report from WordFence [25]. This vulnerability arises from inadequate validation of the ID parameter in the plugin function that stores a new or a modified customer object. Although the function verifies user capabilities, making only certain roles allowed to call the function, it fails to restrict the scope of objects that can be modified. As a result, an authenticated attacker with lower privileges, such as managers, can exploit this flaw to alter the email addresses of administrator accounts by providing valid administrator IDs (see Figure 1). After recording her email address, the attacker can call password resets, gain unauthorized access to administrative accounts, and potentially take full control of the affected WordPress site.

Another example is CVE-2024-7437 [6], which happens in the SMF (Simple Machines Forum) application. In this application, BAC occurs when a malicious user alters the vulnerable parameter (i.e., *alert ID*) in the request, allowing him to delete other users' alerts. This vulnerability is similar to the one in WordPress, as improper authorization checks on client-submitted object references cause both.

3 Preliminary Analysis of BAC Characteristics

To gain deeper insights into the characteristics and distribution of BAC-related vulnerabilities in real-world software systems, we performed a preliminary investigation of the BAC attributes associated with publicly disclosed CVEs from recent years.

3.1 Data Collection

Firstly, as explained in Section 2.1, we use *Broken Access Control* and *IDOR* as the primary keywords in our search to query the CVE repository [24] for reports published in the last three years (between April 2022 and April 2025). We further restricted our CVE selection to cases that occurred in open-source web applications. This constraint ensures that we can reproduce the vulnerable behaviour in a local environment and inspect the source code if necessary. Projects that required commercial licenses, closed-source systems, or non-web contexts were excluded. In addition, since the grey-box web fuzzer we design requires platform-specific instrumentation, we limit our focus to CVEs affecting PHP-based applications, as PHP remains one of the most widely used platforms. Finally, to ensure the presence of BAC vulnerability, we manually reviewed each CVE entry and included only those with sufficient technical detail (e.g., vulnerable URL and parameters) to reproduce the issue.

3.2 Challenges to Reveal BAC

We collected 15 CVEs (see Table 1) after applying the methodology described in the previous section. We then manually analyzed the vulnerable functions and objects to find ways of triggering these vulnerabilities through the web UI with the aim of identifying relevant HTTP requests to submit and verifying successful trigger/exploitation. Since our goal is to design a fuzzer tailored to BAC vulnerabilities, we identified the challenges for automation of the process.

C1: Identifying Potentially Vulnerable Functions. Web applications dynamically generate function calls based on user context, session state, or permissions, making the discovery process heavily context-dependent. Furthermore, they may rely on aliasing mechanisms in their APIs, making the server file names irrelevant to available endpoints. Consequently, traditional methods, such as scanning for files on the server and invoking them through HTTP requests can be less effective.

Consider CVE-2023-43663 (see Table 1), which affects the PrestaShop application (has 8.6k stars on Github) [37], as an example of this observation. The vulnerable function of the Prestashop application lies in the *AdminDashboardController.php* file, which is not visible in the URL. To trigger that function, rather than calling the file name via URL, a non-admin user should call a certain URL ending with */disable/«module_id»*. Failing to produce the correct URL with the correct *module_id* prevents the fuzzer from reaching the vulnerable function.

In addition to collecting all functions, flagging some of the functions as potentially vulnerable is also challenging. Popular security tools, such as OWASP ZAP [32] and Burp Suite [36], solve this challenge by requiring users to manually configure an access control list (ACL) that defines which functions should be allowed or denied for specific users. However, such reliance on human intervention undermines the applicability of that solution for fuzzing, which aims to minimize manual effort.

C2: Identifying Potentially Vulnerable Objects. Differentiation of allowed and restricted objects is crucial to precisely attempt to trigger BOLA vulnerabilities. Due to the context-dependent nature of the objects, opening a webpage with a specific user role and observing visible objects provides a baseline for identifying resources accessible to that role. However, the real challenge lies in the discovery of restricted objects, particularly those with differentiated access permissions (e.g., accessible for READ operations but restricted for UPDATE or DELETE actions). In addition, the restricted objects are frequently hidden behind indirect references or system-generated values that are only visible in the body of the HTTP request and therefore are difficult to observe without manual inspection.

For example, in CVE-2024-8290 and CVE-2024-7437, restricted objects are related to the administrator and alert, and are referred using *admin ID* and *alert ID*, respectively. Failing to identify the parameter names and generate the correct IDs prevent the web fuzzers from triggering the vulnerability.

C3: Oracle Verification. Semantic bugs like BAC do not easily manifest themselves in observable states (e.g., crash, memory corruption, etc). Therefore, confirming the occurrence of the BAC vulnerability is hard because there is no clear signal from WUT to confirm, especially in the case of unexpected resource modification. Solely relying on web response messages can be less effective since the WUTs may reply with unclear information.

For example, in CVE-2025-3536, when a malicious user calls the vulnerable URL with a correct *user_id*, the server replies with redirected responses without a clear success message. Another example, in CVE-2024-8290, when calling the vulnerable URL with an unexisting *admin_id*, a user receives a clear success message even though the server does not execute the update admin request due to

Table 1: Collection of CVEs reporting BAC, sorted by affected App size (the number of LoC). For simplicity, HTTP Param refers to name-value pairs placed in the HTTP header, the URL query string, or the HTTP body.

CVE No.	Affected App (+ Plugin)	Method	Vulnerable URL	Vulnerable Param
CVE-2025-0843	Library Card System	GET	/del.php	del=«student_id»
CVE-2025-3536	Employee Management System	GET	/admin/delete-user.php	id=«user_id»
CVE-2025-3537	Employee Management System	POST	/admin/update-user.php	user_id=«user_id»
CVE-2024-55231	Notes Sharing Management System	POST	/user/edit-notes.php	editid=«id»
CVE-2024-55232	Notes Sharing Management System	GET	/user/manage-notes.php	delid=«id»
CVE-2024-40480	Online Exam System	GET	/admin/update.php	uemail=«email»
CVE-2025-0802	Best Employee Management System	POST	/admin/Operation/User.php	del_id=«user_id»
CVE-2023-46449	Inventory Management System	POST	../action/edit_update.php	user_id=«user_id»
CVE-2024-3139	Computer Lab Management System	POST	/classes/Users.php	f=save id=«id»
CVE-2024-9082	Online Eyewear Shop	POST	/classes/Users.php	f=save id=«id»
CVE-2024-7658	Projectsend	GET	/process.php	do=get_preview&file_id=«id»
CVE-2024-7437	Simple Machines Forum	GET	/index.php	do=remove;aid=«alert_id»
CVE-2024-7438	Simple Machines Forum	GET	/index.php	do=read;aid=«alert_id»
CVE-2024-8290	WordPress + WC + WCFM	POST	/wp-admin/admin-ajax.php	customer_id=«admin_id»
CVE-2023-43663	Prestashop	GET	.../action/disable/«module_id»	

the incorrect *id*. Therefore, the fuzzer needs accurate information that cannot be obtained from the response messages.

C4: Reducing Randomness. In general, web applications expose a large number of endpoints with numerous parameters in the request headers, URL and body. This situation makes random selection of one of these inputs and its alteration with a byte array or random strings less effective because there are huge input spaces to explore. Normally, most web servers apply certain input validation rules, making them often reject abnormal inputs. To make fuzzing more effective, certain endpoints and parameters that have a greater chance of triggering the access control vulnerability should be prioritized. For example, in CVE-2024-8290, mutating the value of the *name* parameter is unlikely to trigger a BAC vulnerability, whereas mutating the *customer_id* parameter does.

In addition, web fuzzers should reduce the use of completely random values in order to increase chances of reaching and triggering vulnerable code. For example, in CVE-2024-40480, filling the *uemail* with a random value or a grammatically valid value that does not exist in the database will not trigger BAC. Therefore, to work efficiently in revealing BAC, a fuzzer should use fewer random values.

3.3 Scope of the Work

Based on the insights gained from the preliminary analysis, we define the scope and assumptions that underlie the proposed approach.

(1) Given the wide range of BAC cases, our work focuses only on role-based access control (RBAC) [40], arguably one of the most commonly implemented access control models, which allows or prohibits certain users from accessing or modifying any function or object based on their roles. Other models, such as context-based or attribute-based access control [41], which regulate whether users are allowed to access certain objects based on the state of the users, are out of the scope.
(2) Our work is limited to uncovering BAC at the code level. Issues caused by design-time errors (e.g., a user with limited privileges is assigned a role beyond her privileges) are out of the scope. We assume that such errors do not reflect a vulnerability in the code but in the instantiation of the RBAC model (i.e., user-role and role-permission assignments).
(3) The proposed fuzzer does not explore available actions to insert data in the beginning. We assume the human tester normally prepares the WUT with some initial data, including registered users with different roles, before performing security testing.

4 Proposed Approach

Based on the challenges described in Section 3.2, we propose three techniques tailored to BAC vulnerability detection: hierarchical role analysis, reference mutation, and SQL checking. These techniques are then implemented in a fuzzer we call **BACFuzz** (Section 5) in revealing BAC.

4.1 Hierarchical Role Analysis

For vulnerable function identification (C1), we propose hierarchical role analysis to find HTTP requests that only appear in higher-role users. Firstly, this process makes the fuzzer open web pages with a specific user role, navigate the pages, fill forms, click buttons and links, and save HTTP requests the browser sends. The fuzzer repeats these actions for all available user roles. All saved requests replied to with a valid response code (i.e., 2xx) are stored in the **request corpus** grouped by their roles. Furthermore, param-value pairs are extracted from URL queries and body payloads and stored in another corpus (i.e., **param corpus**). To determine whether a request is new and unique, the fuzzer compares the request URL, request method, URL queries (only param name without value), and body payloads (only param name without value).

The request that appears in certain user roles yet disappears in the other roles is marked as a potentially vulnerable function. Then, the fuzzer can focus on **BFLA testing** (also called *vertical access control testing*) on these request types by sending the requests using a lower-role account. As explained in Section 3.2, the vulnerable function in CVE-2023-43663 can be triggered by calling the link using a lower-role account. For the other requests that are not marked, the fuzzer conducts **BOLA testing** (also called *horizontal access control testing*) by applying reference mutation (Section 4.2).

***Role Labelling*.** The fuzzer labels each request with role names that the fuzzer uses to trigger the request. For example, the fuzzer logging in with an admin account can execute some actions on the admin page. The requests collected during this session are labelled with admin. When the same request comes from another fuzzer instance logging in with a different role (e.g., manager), the fuzzer only adds a new label (i.e., manager) to the request in the corpus. In addition, the fuzzer does the same labelling process for param-value pairs stored in the param corpus.

4.2 Reference Mutation

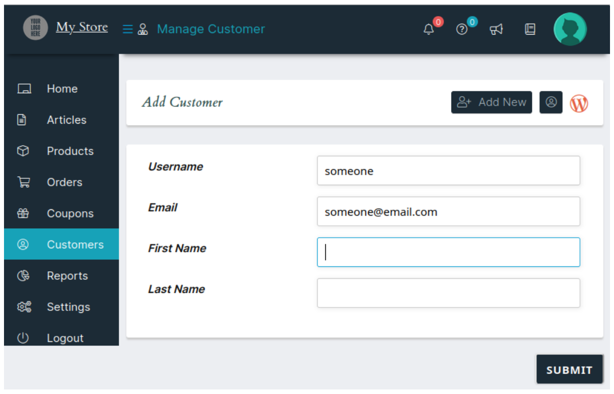
To trigger BOLA vulnerabilities from collected requests, we propose a reference mutation. This proposed method involves sys-gen data collection, reference parameter analysis, and value alteration.

4.2.1 Sys-Gen Data. For vulnerable object identification (C2), we first observe system-generated (sys-gen) data, which is restricted data that is invisible in users' direct view but visible in generated HTTP requests. Compared to user-supplied input that the web users can fully control, the sys-gen data is not controllable from the web UI. As explained in our analysis (Section 3.2), BAC occurrences come from unexpected alterations in sys-gen data; thus, manipulating it can trigger BAC. In addition, OWASP released the top 25 vulnerable parameters [31], and all of them are in the form of sys-gen data, making the sys-gen data an ideal mutation target. Web developers commonly place the sys-gen data in HTML files (e.g., in a hidden tag) or JavaScript files (e.g., in a function triggered on form submission). For example, Figure 2 demonstrates a filled-in form and an HTTP request generated by the web browser, in which there are additional (sys-gen) fields in the request that are invisible from the user's view.

***Data Collection*.** To collect sys-gen data, we analyze the intercepted requests stored in the corpus. Each key-value pair in the request URL or request body is classified as either the user-generated or sys-gen group by checking the value part. Since the fuzzer fills in all HTML forms with random values concatenated with certain pre-defined values, if a parameter value contains the pre-defined value, it is user-generated data; otherwise, it is sys-gen data.

4.2.2 Reference Parameter Analysis by LLM. Since not all sys-gen data represents web functions or objects, we utilize LLM to further find out which sys-gen data refers to a function or an object. Therefore, after marking sys-gen parameters, the fuzzer queries LLM with a prompt described in Figure 3. Getting replies with parameter names that may refer to functions or objects from LLM, the fuzzer marks them as **reference params** and special mutations are prepared for them. The other parameters are marked as less important (see Figure 4), which means random mutation is always applied to them. According to their values or the name matching the predefined rules, parameters can be marked as security measures, which means the fuzzer will never select them for mutation.

4.2.3 Mutation. To reduce randomness (C4), the fuzzer should more often use values from the param corpus collection rather than



Username	someone
Email	someone@email.com
First Name	
Last Name	

(a) Add customer form

```
action: "wcfm_ajax_controller"
controller: "wcfm-customers-manage"
wcfm_customers_manage_form: "user_name=someone&user_email=someone%40ema
                            il.com&first_name=&last_name=&customer_id=0&bfir
                            st_name=&blast_name=&bcompany_name=&bphone
                            =&baddr_1=&baddr_2=&bcountry=&bcity=&bstate=&
                            bzip=&same_as_billing=yes&sfirst_name=&slast_nam
                            e=&scompany_name=&saddr_1=&saddr_2=&scountry
                            =&scity=&sstate=&szip=&wcfm_nonce=0b1f003038"
status: "submit"
wcfm_ajax_nonce: "f20a369de5"
```

(b) HTTP Request Body

Figure 2: In CVE-2024-8290, after a user fills in a form (2a) and clicks the submit button, the browser generates an HTTP request with body payload (2b) containing additional fields called sys-gen data: *action*, *controller*, *status*, and *wcfm_ajax_nonce*, which is invisible in users' direct view. Furthermore, the field of *wcfm_customers_manage_form* also consists of some sys-gen data.

random values for request mutation. Basically, reference values are grouped into two types: numeric and text, in which the former is any value starting with a number (can be followed by text) and the latter is otherwise. When the fuzzer takes an HTTP request and selects one of the reference parameters in the request for mutation, the value of the selected reference parameter is changed to one of the other reference values of the same type. It aims to reduce server rejection and increase mutation effectiveness since the reference values, especially the numeric ones, are usually limited and may refer to a certain object reference that has been seen before.

Random Mutation. A random mutation that produces random values is still used and commonly applied to less-important parameters. When selecting an HTTP request for mutation, the fuzzer may only mutate selected reference parameters or also involve random mutation for selected less-important parameters.

4.3 SQL Checking

For Oracle verification (C3), we propose SQL checking, which verifies whether WUTs execute the access attempt to protected references. While SQL queries alone may not be definitive proof of existence bugs, arbitrary values in data-manipulation queries indicate improper authorization checks. Illustrated in Figure 5, generally, a WUT generates DML (data manipulation language) to be sent

# Overview
You are a web fuzzer trying to reveal broken access control problems on web functions and objects. I will give you one HTTP request consisting of the request method, URL, and body payload I have collected. Based on this request, your goal is to find certain parameter names or body payload names that might refer to restricted objects. Then, I will mutate the names you will give to examine if the mutated values for the names can access the restricted objects.

# Collected Requests
I have crawled the web under test and caught many HTTP requests. Now, I am giving you one request with a format of [method]<space>[full_url]<space>[body_payload].

# Your Task
Based on that HTTP request, you have to point out which parameters or payload names that could potentially refer to restricted objects, excluding purely data-related fields or security parameters (e.g., nonces or security token). Also, you have to point out which parameters or payload names that are reference IDs. You might need to break down nested parameter values into their atomic components and only list their atomic parameters or payloads. Please note, only answer the question without explanation. Only use comma to separate the parameter or payload names.

Figure 3: LLM Prompt for Parameter Analysis.

```
POST /wp-admin/admin-ajax.php
Body Payload:
  _nonce: 72f9a3c68d
  action: heartbeat
  has_focus: 'true'
  interval: '60'
  screen_id: dashboard

----
POST /wp-admin/admin-ajax.php
Body Payload:
  action: wcfm_generate_variation_attributes
  wcfm_ajax_nonce: 3f31cf9eb7
  wcfm_products_manage_form:
    product_type=simple
    stock_qty=0
    ...
```

```
POST /wp-admin/admin-ajax.php
Body Payload:
  _nonce: <<security_measure>>
  action: <<reference_param>>
  has_focus: <<less_important>>
  interval: <<less_important>>
  screen_id: <<reference_param>>

----
POST /wp-admin/admin-ajax.php
Body Payload:
  action: <<reference_param>>
  wcfm_ajax_nonce: <<security_measure>>
  wcfm_products_manage_form:
    product_type=<<less_important>>
    stock_qty=<<less_important>>
    ...
```

Figure 4: Two examples of HTTP requests after parameter analysis. The requests on the left are the original, and the right are the results.

to the DBMS when executing a request. To enable the real-time check, this study instruments the web interpreter to record all SQL queries the WUT generates. Then, the proposed fuzzer can check if the mutated values appear in the query.

4.3.1 *Checking Rules.*

Several rules are determined to automatically infer the occurrence of the BAC.

Rule 1: Broken Function-level. Based on the BFLA definition, a web function is called vulnerable to access control if the function that only exists in certain roles is successfully executed by users operating under different roles. Therefore, when the fuzzer submits a request using a lower-role account and detects a DML query, it is **BFLA** if: 1) at least one value in the query is the same as the parameter value of the submitted request, and 2) the request does not exist in the user page. To ensure the second requirement is satisfied, the fuzzer opens the *referer* link stated in the request header to check if the trigger source (e.g., button or link that may trigger the request) or the parameter values exist in the page.

# Submitted HTTP Request
POST /lms/classes/Users.php?f=save

id=**1711440657**&firstname=ibdjweyxfoz&middlename=wnmnozap ....

# Generated SQL Query
UPDATE users set firstname = 'ibdjweyxfoz' , middlename = 'wnmnozap' , lastname = 'swkgxicagowetfbp' , username = 'tzqwyaloxx' , password = 'c82a88d7c5f180798b0d118d23893a94' where id = ***1711440657***

Figure 5: When a database-backed web app executes a request, it commonly generates an SQL query corresponding to the submitted data. For example, in CVE-2024-3139, the value of 1711440657 in the request payload appears in the SQL query. Therefore, the occurrence of certain values in an SQL query can be a sign that a certain request is executed, not rejected.

Rule 2: Broken Object-level. Based on the BOLA definition, an object is called vulnerable to access control if the object reference that only exists in certain roles is successfully accessed by users operating under different roles. The functions or URLs may exist in all roles, but they manipulate different objects. Therefore, when the fuzzer submits a request with altered reference params and detects a DML query, it is **BOLA** if: 1) at least one value in the *WHERE* clause is the same as one of the mutated reference values in the corresponding request, and 2) the mutated reference value is not found in the corpus of the role that is used to submit the request. Due to the dynamic nature of the object, to ensure the second condition is satisfied, the fuzzer opens the *referer* link stated in the request header, extracts the reference parameters from the opened page, and checks if the mutated values do not exist in the extracted parameter values.

4.3.2 *Multiple Checking for BAC Validity.*

Using the SQL checking method, the fuzzer can detect a BFLA or BOLA; however, the fuzzer still needs to apply multiple checks to ensure that the values appearing in the SQL query are not by accident and indeed related to the mutated values in the submitted request. So, when the checking method marks a request as BAC according to the aforementioned rules, the fuzzer selects and mutates the request again. If the SQL checking detects the same BAC after the WUT executes the mutated request, then **the BAC is valid**. To increase the confidence level of BAC detection, the fuzzer selects and mutates the request again and again until the SQL checks do not raise BAC or the checking repetition reaches a certain number (e.g., 10 times repetition).

5 BACFuzz: Fuzzer Implementation

To adopt the proposed approaches, this study designs a new grey-box web fuzzer called **BACFuzz**.

5.1 Overview

There are two main components of **BACFuzz**: the *main driver* and the *active checker.* While the main driver navigates whole web pages in the WUT and stores submitted HTTP requests, the active

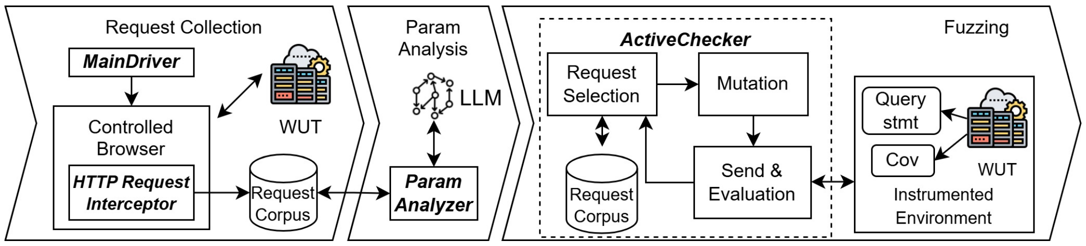


Figure 6: Proposed BACFuzz pipeline contains three main phases: request collection, parameter analysis, and fuzzing.

Algorithm 1 BACFuzz

```
Require: $baseUrl$
Require: $Roles$                                  ▷ list of available roles
 1: for $r$ in $Roles$ do
 2:     $state_r \leftarrow userLogin(baseUrl, r)$
 3:     $d \leftarrow$ new $MainDriver(state_r, r)$
 4:     async $d.crawl()$          ▷ start the driver to navigate pages
 5: end for
 6: $target \leftarrow getHighestRole(Roles)$              ▷ e.g., Admin
 7: $Roles \leftarrow excludeTarget(target, Roles)$
 8: $resetWUT()$
 9: for $r$ in $Roles$ do
10:     $state_r \leftarrow userLogin(baseUrl, r)$
11:     $c \leftarrow$ new $ActiveChecker(state_r, r)$
12:     async $c.fuzz()$                 ▷ start the active checker
13: end for
```

checker processes the stored requests to exploit any BAC possibility in the WUT. Following the best practice of Witcher [44], we run both components sequentially, in which the main driver runs first.

In general, our proposed fuzzer (see Figure 6) works in the following order: (1) collecting web functionalities and objects from each user role; (2) analysing reference parameters; and (3) altering the parameters and verifying the vulnerability occurrence. Initially, the fuzzer user sets the WUT URL and a list of registered accounts with various roles, including an anonymous account (see Algorithm 1). Iterating the roles, the fuzzer opens the URL and requires the user to open the login page and input the account credentials (line 4). After that, the driver works automatically to navigate whole web pages using the logged-in credentials (line 6). This process will be explained further in Section 5.2. Then, the fuzzer determines the highest user role as the target. The active checker performs fuzzing using each user role (excluding the target), which will be explained further in Section 5.3.

5.2 Main Driver

The main driver collects HTTP requests by navigating whole web pages using each user role. To enable the request collection, there are two main procedures in the main driver: *Drive* and *InterceptRequest*. The first procedure navigates the entire web page to find HTML pages manipulate web resources. The procedure looks for pages that have the *form* tag and its derivatives (e.g., the *input*, *select*, and *textarea* tags) because those allow users to input something to be sent to the server. On the other hand, the *InterceptRequest* procedure aims to catch HTTP requests and store them in the request corpus. In addition, key-value parameters, in the request URL and body, are stored in the param corpus. We attached this procedure to the browser manipulation library, guaranteeing it is invoked whenever the browser sends any HTTP request.

5.2.1 User Login. To explore the whole web application functions, the main driver requires the fuzzer user to log in using some accounts with different roles. For example, if the user intends to test WordPress, which consists of five roles: administrator, editor, author, contributor, and subscriber, the driver will prompt the user to log in five times. This login process can be done automatically, which requires the user to configure login scripts to align with the WUT page. After storing all cookies and the user URLs (dashboard pages) [27], the fuzzer instantiates some driver instances to navigate the main page using the cookies.

5.2.2 Crawling Strategies. According to the work of Stafeev and Pellegrino [42], which reviews many crawling algorithms, there are two crucial problems that need to be specified when designing a web crawler: navigation and page similarity.

Navigation strategy. Similar to the work of Khodayari et al. [14] which crawls around 1M web pages, our main driver visits web pages with a depth-first strategy, which means visiting the most recently discovered link. Given the homepage of the logged-in user, the driver identifies web links by looking for anchor tags and saves them in the *links* variable.

Page similarity. The proposed driver uses URL components as the objects for page similarity, which means only visiting pages with different URL components. When finding a web link from an anchor tag, the driver takes its visible text and URL components (i.e., base URL, URL path and query string containing sys-gen data) and compares this information with previously saved links. If it is non-identical, the found link is stored and will be visited.

5.3 Active Checker

The main idea of the active checker is to alter the captured HTTP requests to reveal BAC. In general, the active checker chooses one HTTP request from the seed (i.e., request corpus), performs one of the mutation strategies, sends the mutated requests to the WUT using a certain user session, and checks the feedback.

5.3.1 Request Selection. The active checker prioritises HTTP requests with more reference parameters to be selected because the endpoints of those requests have more attack surfaces that can be exploited. To implement that idea, the request selection process involves a random function with the number of reference params as the weight.

5.3.2 Mutation. After obtaining a request for mutation, the active checker chooses what kind of testing (either function- or object-level) to conduct based on the request mark (explained in Section 4.1).

Function-level Testing. BFLA happens when the WUT execute HTTP requests that are unavailable for certain user roles. For example, the request of *add_item* that only exists on the administrator page will be tested using a non-admin account. So, when taking an HTTP request having a different label from the checker name (explained in Section 4.2.1), the active checker will perform such testing because the request is supposed to be rejected by the WUT due to forbidden access. To be more effective, the active checker adjusts the chosen request with unique data from the selected accounts, such as *nonce* data that is unique per account in WordPress.

Object-level Testing. BOLA mostly happens when a certain role is allowed to execute certain functions but the WUT does not validate the object the user tries to modify. Therefore, to test the object level, the checker takes one request with the same label as the checker's logged-in role and mutates the selected parameters by using another object reference value (Section 4.2.3).

5.3.3 Feedback Evaluation. The feedback the active checker receives from the WUT (i.e., HTTP response code, response messages, code coverage, and SQL queries) is evaluated to decide whether the sent request has triggered BAC. As explained in Section 4.3, the active checker compares parameter data inserted in the submitted requests with the SQL query the WUT produces. If the data exists, the request and the query are reported to the user and added to the attack surface collection. In addition, when requests fail to trigger BAC yet bring new code coverages or 500-response codes, the active checker also puts these requests into the attack surface collection to be explored further. As shown in a previous study [45], these interesting requests can guide the fuzzer to reach deeper statements in WUTs. We count the 500-response code as well because some studies [5][10] stated that this code is useful to show web defects and can lead to more vulnerabilities.

5.4 Instrumentation

Instrumentation is critical in grey-box fuzzer setup because it produces live and lightweight internal information [21]. This study follows the work of Neef et al [26] that instruments web applications by using the *function hooking*, a feature provided by the UOPZ library [34], to collect SQL queries sent by WUT. Using this technique enables the instrumentation scripts to manipulate original PHP functions related to SQL calls, such as *mysqli_query*, *mysqli_stmt_prepare*, *mysqli_prepare*, and *PDOStatement* with additional codes acting as query catching. In addition to the query collection, we use the PCOV library [47] for coverage accounting. Since the library has a better performance overhead than Xdebug, it helps generate accurate line coverage reports quickly [4]. Since each request is supplied with a unique identifier (named *X-FUZZER-COVID*) attached in the header, both query and coverage information are written in JSON files named with each corresponding request identifier.

5.5 Counting Unique Results

After running a certain of time, **BACFuzz** reports the final results to the user. To count unique BAC cases, **BACFuzz** compares the submitted request URL and method with the detected SQL query. For example, CVE-2024-7437 and CVE-2024-7438 use the same URL and method to trigger the vulnerability; however, the SQL queries produced are different (i.e., *DELETE FROM smf_user_alerts* vs *UPDATE smf_user_alerts*), making them two different cases.

5.6 Implementation

We implemented the proposed fuzzer in Python because it has libraries that fit our needs, such as Playwright [23] for the browser controller and request interceptor, and BeautifulSoup [39] for HTML processing. Then, we implemented the instrumentation, including the SQL checking, in PHP scripts with the UOPZ and PCOV libraries. To ease the evaluation process, we deploy the WUTs and evaluation scripts in Docker because handling complex web server components is straightforward with Docker. In addition, we use Docker to make it easy for other researchers to duplicate our study.

6 Empirical Evaluation

In order to evaluate the effectiveness of **BACFuzz** comprehensively, we run it against test-bed and real web applications with known and unknown access control vulnerabilities. At the end of the experiments, we aim to answer the following research questions.

- RQ1. Can **BACFuzz** report the known vulnerabilities?
- RQ2. Can **BACFuzz** uncover new and valid vulnerabilities?
- RQ3. How much overhead does **BACFuzz** introduce?
- RQ4. How effective is **BACFuzz** in generating tests?

6.1 WUT Collection

As WUTs with known vulnerabilities, we use web applications affected by 15 CVEs summarized in Section 3.2. We also include benchmark applications employed by recent studies [18][26][12] that are PHP-based: *Damn Vulnerable Web Application* (DVWA) and *Xtreme Vulnerable Web Application* (XVWA), since they provide BAC vulnerabilities for training purposes.

Recently, a static analysis work [13] revealed BOLA vulnerabilities in 25 web applications. Even though this work successfully detects many vulnerabilities in most tested applications, it finds nothing in four applications: Scarf, PhpBB, Opencart, and Zencart. Except for Scarf, which is no longer updated, we also evaluate our work on those three remaining applications to demonstrate that our fuzzer can reveal vulnerabilities that the static analysis work cannot detect.

6.2 Experimental Setup

For each WUT, we run the **BACFuzz** main driver for a maximum of 24 hours to collect all HTTP requests and continue to run the active checker for 24 hours to reveal BAC from the requests. We run the experiments in a virtual computer with 8 CPUs and 32 GB RAM and use the *DeepSeek-V3* model provided by *DeepSeek* [8] for LLM

Table 2: Evaluation results sorted by the number of WUT LoC. It shows that BACFuzz can detect 16 out of 17 known cases and reveal 26 new TP BAC. The only failed detection occurred in CVE-2023-43663 due to discrepant submitted values.

App Name / CVE No	Available Roles (apart from Admin and Anonymous)	Number of			Req Col. Time	Known BAC	Detect Time	New BAC Detected		Avg. Resp. Time	Instr. Over head	Non-Rej. Req.
		Files	LoC	Req				TP	FP			
DVWA	User	0.2K	7.5K	71	0:04:15	BFLA	0:00:02	0	0	0.025	0.001	99%
XVWA	User	0.7K	17K	44	0:26:33	BFLA	0:00:15	0	0	0.026	0.003	100%
CVE-2025-0843	User	17	0.5K	13	0:01:19	BFLA	0:00:02	1	0	0.021	0.003	100%
CVE-2025-3536	Employee	40	1.5K	63	0:04:42	BFLA	0:00:02	1	0	0.023	0.003	100%
CVE-2025-3537						BFLA	0:01:25					
CVE-2024-55231	User	0.3K	2.4K	38	0:02:33	BOLA	0:00:06	0	0	0.022	0.004	99%
CVE-2024-55232						BOLA	0:00:28					
CVE-2024-40480	User	0.1K	4K	78	0:06:24	BFLA	0:00:06	2	0	0.024	0.007	99%
CVE-2025-0802	LeaveMan., Salary, BorrowMan.	0.6K	23K	124	0:15:37	BFLA	0:01:54	3	0	0.022	0.002	100%
CVE-2023-46449	Staff	1.1K	145K	67	0:12:14	BOLA	0:01:36	8	0	0.021	0.003	100%
CVE-2024-3139	Staff	2.5K	149K	28	0:03:12	BFLA	0:00:08	2	0	0.018	0.001	99%
CVE-2024-9082	Staff	2.6K	152K	71	0:18:13	BFLA	0:00:52	2	0	0.024	0.002	100%
CVE-2024-7658	Manager, Uploader, Client	12.3K	159K	642	3:05:02	BFLA	0:0:28	0	0	0.091	0.039	61%
CVE-2024-7437	User	0.8K	188K	1458	2:49:37	BOLA	0:06:46	1	5	0.092	0.041	73%
CVE-2024-7438						BOLA	0:04:21					
CVE-2024-8290	ShopMan., Author, Subscriber, Cust.	9.3K	929K	548	3:10:04	BOLA	0:03:06	4	2	1.423	0.864	54%
CVE-2023-43663	Logistician, Translator, Sales, Cust.	29.3K	2.4M	1897	4:09:38	BFLA	X	1	3	0.298	0.162	99%
OpenCart	Cataloger, Marketing, Customer	9.8K	434K	288	0:30:34	-	-	0	1	0.105	0.025	55%
ZenCart	Order Processor, Customer	9.7K	585K	878	2:37:27	-	-	1	0	0.476	0.131	91%
phpBB	Member	7.8K	726K	543	1:10:39	-	-	0	0	0.101	0.052	97%

Req Col Time = Time to collect all unique HTTP requests; Avg Resp Time = Average time of WUT, which has been instrumented, to reply; Instr Overhead = Time overhead caused by instrumentation; Non-Rej Req = Proportion of requests that are not rejected by WUT.

service because it has good performance in the code generation domain. Since each fuzzing campaign works randomly, making the result not the same in each experiment, we run the fuzzer three times and report the average results.

In the experiments, we do not compare our work with other popular security tools, such as OWASP ZAP and Burp Suite, due to the significant difference between functionality and scope. Unlike our approach, they are not designed for fully automated, object-level access control testing, making them not applicable for a direct comparison. In addition to those popular tools, there are several scientific works addressing BAC, such as [38][16][43] (explained more in Section 8); however, the source codes of these works are not available, preventing us from comparison. There is also a static analysis work [13] to reveal BOLA vulnerabilities, but the authors confirmed that it is challenging to manually create *dal_specification.json*, the required file for testing new WUTs.

6.3 Experiment Results

Answering RQ1, **BACFuzz** successfully reported 16 out of 17 known vulnerabilities (see Table 2). The only failed detection happens in the CVE-2023-43663 case due to discrepant submitted values. In that case, which affects Prestashop, when the disable module function is called, the WUT converts the module ID to a corresponding primary-key ID in the DBMS and then sends the converted ID through a SQL query. For example, **BACFuzz** sends *statsbestcustomers* in the vulnerable parameter, but the WUT sends the query of "*UPDATE ps_module SET active = 0 WHERE id_module = 64*" to DBMS, making the fuzzer unable to detect the matched value.

Answering RQ2, **BACFuzz** successfully reported new vulnerabilities with low false positive rates. **BACFuzz** also reported one new BAC from applications that the static analysis [13] could not detect. We reported these vulnerabilities to the developers of the respective applications and are waiting for the developers' responses to confirm our findings. If we receive confirmation before the final version deadline, we will include their responses in the revision. Otherwise, we will clarify the disclosure timeline and our communication efforts in the paper.

To answer RQ3, we compare the time WUTs need to reply to each request between those using and not using instrumentation. We deactivate the instrumentation by not putting the *X-FUZZER-COVID* header in the request (as explained in Section 5.4). Table 2 shows that the use of instrumentation for coverage and query collection results in less overhead.

To answer RQ4, we compare the number of not-rejected requests (replied with either 200 or 500 response codes) because those requests are interesting, as explained in Section 5.3.3. The results in Table 2 show that the proposed mutation strategies can produce a large proportion of non-rejected requests, which is good for exploiting various logic functions in WUT.

6.4 Discussion: False Positive Result

Although SQL query can serve as an oracle for detecting BAC, its application may also result in false positive (FP) results. Based on our observation of evaluation results, FP happens because of the coincidence of the same value and the nature of dynamic objects.

First, the matched value between the mutated parameter and the captured SQL query is syntactically valid yet semantically incorrect. For example, in OpenCart, **BACFuzz** collects a query of "*DELETE FROM oc_cart WHERE (api_id > 0 OR customer_id = 1)*" after sending a request with *route=0* as the mutated parameter value. Since this condition is marked as BAC due to the presence of 0 value, multiple checks are performed by sending a new mutated request with *route=1*. Once again, the condition is marked as BAC because the value 1 exists in the collected query. However, we can obviously see that the submitted request and the obtained query are not correlated.

Second, dynamic object behaviour causes FP because certain objects are missing during crawling, but then exist during fuzzing. For example, in SMF, a user had only one readable topic on his page initially, making the MainDriver store the *id_topic* 1 as the available object for the user. However, during the fuzzing campaign, the ActiveChecker might send requests that lead to another topic creation (e.g., *id_topic* 2) for the user. Therefore, because *id_topic* 2 is still considered inaccessible but **BACFuzz** successfully modifies it (because it is indeed accessible for the user), **BACFuzz** raises a BOLA flag that is FP.

7 Threats to Validity

While this study demonstrates the effectiveness of **BACFuzz** in revealing BAC vulnerabilities, some threats to validity must be considered when interpreting the results.

Internal Validity. The internal threat arises from how large WUTs, such as SMF, WordPress, and PrestaShop, handle logging, as they store extensive runtime events in dedicated log tables. These log tables commonly record errors and unauthorized access attempts, making SQL queries mostly match the submitted requests. Therefore, it is suggested to exclude this kind of table that always stores all submitted requests. To make the fuzzer only report valid vulnerabilities, the fuzzer user should put these ignored tables in the **BACFuzz** configuration. These ignored tables are crucial to maintaining the report's precision. In our experiment, we exclude tables of *smf_log* (CVE-2024-7437 and CVE-2024-7438) and *ps_connections* (CVE-2023-43663) from the fuzzer observation. While this filtering step improves the precision of our results, it also introduces a slight dependency on user knowledge of the WUT.

Construct Validity. The construct validity threat comes from our BAC dataset. While additional related CVEs may exist beyond our collection, we believe the selected cases are sufficiently diverse to support the design of our fuzzer. The dataset covers a wide range of BAC patterns and application behaviours, making it a representative basis for guiding and evaluating our design. Nonetheless, our collection methodology may introduce bias toward well-documented or easily reproducible vulnerabilities, which may not capture all real-world scenarios.

8 Related Work

We identified some prior works related to **BACFuzz**, especially in the web fuzzing domain and the access control testing domain.

First, there are some state-of-the-art web fuzzers, such as *EvoMaster* [1], *Restler* [2], *RestTestGen* [46], *bBOXRT* [17], and *RESTest* [22], which work effectively in triggering web crashes. However, they cannot reveal BAC problems because the characteristics between crash and BAC are very different. There are also vulnerability-driven fuzzers which focus more on non-crash vulnerabilities, like *Witcher* [44], *EDEFuzz* [33], *ResolverFuzz* [49], and *Atropos* [12]; however, they are not revealing BAC.

Second, there are some recent works on access control testing. Rennhard et al. [38] and Kushnir et al. [16] developed practical solutions to automatically detect BAC on web applications. Since their works are limited by manipulating GET requests only, performed in a black-box setup, and comparing web responses for vulnerability verification, these are different from our work. Our proposed fuzzer manipulates more attack vectors, works in a grey-box setting with more advanced techniques, and uses SQL queries to verify the vulnerability. The work of Sun et al. [43] used static analysis for vulnerability detection; however, it only works well for traditional web applications that are not AJAX-heavy. Our work uses a dynamic testing setup that catches HTTP requests, rather than observing HTML links, thus it can handle modern web applications. As mentioned in Section 6.2, existing work based on static analysis [13] requires manual creation of test files, making it less practical than our work. Lastly, the work of Kim et al. [15] investigates parameter tampering on the web; however, they highly rely on human intervention to decide whether tampering is successful in altering the business process. Our fuzzer uses automatic verification through query checking, making much less human involvement.

In the domain of vulnerability scanners, there are two widely used tools: OWASP ZAP [32] and Burp Suite [36], which operate in a black-box setting to detect various security issues, including BAC. However, their access control testing is not fully automated. Users are required to manually configure an access control list (ACL) defining which functions should be allowed/denied for specific users. This setup process is both time-consuming and challenging. Furthermore, those tools do not support object-level access control testing and rely solely on web responses to verify ACL violations. As discussed in Section 3.2, this verification method is often insufficient for accurately identifying authorization flaws. To address these limitations, our work introduces automated approaches that eliminate the need for predefined ACLs and accurately verify BAC.

9 Conclusion and Future Work

Broken Access Control (BAC) vulnerabilities remain pervasive in web applications, yet pose unique challenges for automated detection due to the lack of reliable oracles and the difficulty of generating semantically valid attack inputs. In this paper, we presented **BACFuzz**, a grey-box fuzzing framework specifically designed to uncover BAC vulnerabilities—including BOLA and BFLA—by combining hierarchical role analysis, reference mutation, and SQL-based oracle checking. Empirical evaluation across 20 real-world PHP applications demonstrates that **BACFuzz** effectively detects 16 of 17 known issues and uncovers 26 previously unknown vulnerabilities, all with low false positive rates. By releasing the source code and curated evaluation dataset, we aim to foster further research on BAC vulnerabilities and support the development of more secure web applications.

As acknowledged in Section 3.3, there are some types of BAC that fall outside the scope of this work and represent directions for future exploration. First, we identified context-dependent BAC, which refers to vulnerabilities that only manifest after a user performs specific actions, causing a WUT to enter a certain state. These cases require preconditions (e.g., resource creation or feature activation) before unauthorized access becomes observable. Second, we identified passive or view-type BAC, which allows unauthorized users to gain access to sensitive information without modifying any data in the DBMS. Since **BACFuzz** relies on data manipulation (DML) queries to infer unexpected actions, it is difficult to verify the violation of restricted information displayed on user pages that only involve *SELECT* statements. As a result, both context-dependent and passive BAC remain open challenges for future work.

References

[1] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Västerås, Sweden, 394–397. doi:10.1109/ICST.2018.00046

[2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, Canada, 748–758. doi:10.1109/ICSE.2019.00083 ISSN: 1558-1225.

[3] Craig Beaman, Michael Redbourne, J. Darren Mummery, and Saqib Hakak. 2022. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers & Security* 120 (Sept. 2022), 102813. doi:10.1016/j.cose.2022.102813

[4] Sebastian Bergmann. 2024. PCOV or Xdebug? Available at: https://thephp.cc/articles/pcov-or-xdebug?ref=phpunit (Accessed: 2024-12-24).

[5] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 2553–2564. doi:10.1109/ICSE48619.2023.00213

[6] CVE Program. 2024. CVE-2024-7437. Available at: https://www.cve.org/CVERecord?id=CVE-2024-7437 (Accessed: 2024-10-01).

[7] CVE Program. 2024. CVE-2024-8290. Available at: https://www.cve.org/CVERecord?id=CVE-2024-8290 (Accessed: 2024-11-01).

[8] DeepSeek. n.d.. DeepSeek. https://www.deepseek.com/. Accessed: 2025-05-04.

[9] I Putu Arya Dharmaadi, Elias Athanasopoulos, and Fatih Turkmen. 2025. Fuzzing frameworks for server-side web applications: a survey. *International Journal of Information Security* 24, 2 (Feb. 2025), 73. doi:10.1007/s10207-024-00979-w

[10] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented Testing for RESTful APIs. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 739–755. https://www.usenix.org/conference/usenixsecurity24/presentation/du

[11] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology* 33, 1 (Nov. 2023), 27:1–27:41. doi:10.1145/3617175

[12] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 4765–4782. https://www.usenix.org/conference/usenixsecurity24/presentation/gÃijler

[13] Yongheng Huang, Chenghang Shi, Jie Lu, Haofeng Li, Haining Meng, and Lian Li. 2024. Detecting Broken Object-Level Authorization Vulnerabilities in Database-Backed Applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. ACM, Salt Lake City UT USA, 2934–2948. doi:10.1145/3658644.3690227

[14] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 166–184. doi:10.1109/SP54263.2024.00098

[15] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 222–233. doi:10.1145/3377811.3380355

[16] Malte Kushnir, Olivier Favre, Marc Rennhard, Damiano Esposito, and Valentin Zahnd. 2021. Automated Black Box Detection of HTTP GET Request-based Access Control Vulnerabilities in Web Applications:. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy*. SCITEPRESS - Science and Technology Publications, Online, 204–216. doi:10.5220/0010300102040216

[17] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access* 9 (2021), 24738–24754. doi:10.1109/ACCESS.2021.3056505 Conference Name: IEEE Access.

[18] Penghui Li and Mingxue Zhang. 2024. FuzzCache: Optimizing Web Application Fuzzing Through Software-Based Data Cache. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. ACM, Salt Lake City UT USA, 511–524. doi:10.1145/3658644.3670278

[19] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (Sept. 2018), 1199–1218. doi:10.1109/TR.2018.2834476 Conference Name: IEEE Transactions on Reliability.

[20] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[21] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (Nov. 2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[22] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Denmark, 682–685. doi:10.1145/3460319.3469082

[23] Microsoft. 2025. Playwright for Python. Available at: https://playwright.dev/python/ (Accessed: 2025-01-21).

[24] MITRE Corporation. 1999. CVE - Common Vulnerabilities and Exposures. https://www.cve.org/. Accessed: 2025-05-04.

[25] István Márton. 2024. 20,000 WordPress Sites Affected by Privilege Escalation Vulnerability in WCFM – WooCommerce Frontend Manager WordPress Plugin. Available at: https://www.wordfence.com/blog/2024/09/20000-wordpress-sites-affected-by-privilege-escalation-vulnerability-in-wcfm-woocommerce-frontend-manager-wordpress-plugin/ (Accessed: 2024-10-02).

[26] Sebastian Neef, Lorenz Kleissner, and Jean-Pierre Seifert. 2024. What All the PHUZZ Is About: A Coverage-guided Fuzzer for Finding Vulnerabilities in PHP Web Applications. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ACM, Singapore Singapore, 1523–1538. doi:10.1145/3634737.3661137

[27] Eric Olsson, Adam Doupé, Benjamin Eriksson, and Andrei Sabelfeld. 2024. Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 6741–6758. https://www.usenix.org/conference/usenixsecurity24/presentation/olsson

[28] OWASP. 2023. OWASP API Security Top 10. Available at: https://owasp.org/API-Security/editions/2023/en/0x00-header/ (Accessed: 2024-07-05).

[29] OWASP. 2024. OWASP Top 10:2021. Available at: https://owasp.org/Top10/ (Accessed: 2024-10-02).

[30] OWASP. 2024. OWASP Web Security Testing Guide. Available at: https://owasp.org/www-project-web-security-testing-guide/ (Accessed: 2024-10-02).

[31] OWASP. 2025. OWASP Top 25 Parameters. Available at: https://owasp.org/www-project-top-25-parameters/ (Accessed: 2025-01-21).

[32] OWASP ZAP Team. n.d.. The ZAP Homepage. https://www.zaproxy.org/. Accessed: 2025-05-04.

[33] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2024. EDEFuzz: A Web API Fuzzer for Excessive Data Exposures. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3597503.3608133

[34] PHP Documentation Team. n.d.. PHP: uopz Manual. https://www.php.net/manual/en/book.uopz.php. Accessed: 2025-05-04.

[35] PortSwigger. 2024. Access control vulnerabilities and privilege escalation. Available at: https://portswigger.net/web-security/access-control (Accessed: 2024-10-02).

[36] PortSwigger. n.d.. Burp Suite – Application Security Testing Software. https://portswigger.net/burp. Accessed: 2025-05-04.

[37] PrestaShop Contributors. 2025. PrestaShop. https://github.com/PrestaShop/PrestaShop. Accessed: 2025-05-04.

[38] Marc Rennhard, Malte Kushnir, Olivier Favre, Damiano Esposito, and Valentin Zahnd. 2022. Automating the Detection of Access Control Vulnerabilities in Web Applications. *SN Computer Science* 3, 5 (July 2022), 376. doi:10.1007/s42979-022-01271-1

[39] Leonard Richardson. 2025. Beautiful Soup Documentation. https://beautiful-soup-4.readthedocs.io/en/latest/. https://beautiful-soup-4.readthedocs.io/en/latest/ Accessed: 2025-05-04.

[40] Ravi S. Sandhu. 1998. Role-based Access Control. In *Advances in Computers*, Marvin V. Zelkowitz (Ed.). Advances in Computers, Vol. 46. Elsevier, San Diego, CA, USA, 237–286. doi:10.1016/S0065-2458(08)60206-5 ISSN: 0065-2458.

[41] Daniel Servos and Sylvia L. Osborn. 2017. Current Research and Open Problems in Attribute-Based Access Control. *ACM Comput. Surv.* 49, 4, Article 65 (Jan. 2017), 45 pages. doi:10.1145/3007204

[42] Aleksei Stafeev and Giancarlo Pellegrino. 2024. SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 719–737. https://www.usenix.org/conference/usenixsecurity24/presentation/stafeev

[43] Fangqi Sun, Liang Xu, and Zhendong Su. 2021. Static Detection of Access Control Vulnerabilities in Web Applications. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA, 16 pages. https://www.usenix.org/conference/usenix-security-11/static-detection-access-control-vulnerabilities-web-applications

[44] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP) (44)*. IEEE Computer Society, SAN FRANCISCO, 2658–2675. doi:10.1109/SP46215.2023.00007

[45] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webFuzz: Grey-Box Fuzzing for Web Applications. In *Computer Security – ESORICS 2021 (Lecture Notes in Computer Science)*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer International Publishing, Cham, 152–172. doi:10.1007/978-3-030-88418-5_8

[46] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTEST-GEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 142–152. doi:10.1109/ICST46399.2020.00024 ISSN: 2159-4848.

[47] Joe Watkins. 2025. krakjoe/pcov. https://github.com/krakjoe/pcov. Originally posted: 2019-01-16. Accessed: 2025-05-04.

[48] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology* 32, 6 (Nov. 2023), 1–45. doi:10.1145/3597205

[49] Qifan Zhang, Xuesong Bai, Xiang Li, Haixin Duan, Qi Li, and Zhou Li. 2024. RESOLVERFUZZ: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 4729–4746. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-qifan

[50] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (Jan. 2022), 1–36. doi:10.1145/3512345