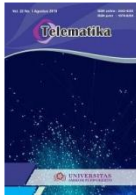

Available online at :
http://ejournal.amikompurwokerto.ac.id/index.php/telematika/

Telematika

Accreditated SINTA "2" Kemenristek/BRIN, No. 85/M/KPT/2020

Performance Analysis of the Fuzzing Method in Detecting API Vulnerabilities in Mobile Healthcare Application X Based on OWASP API Security Top 10

Muhammad Ikhwanul Hakim[1], Radityo Adi Nugroho[2,*], Dodon Turianto Nugrahadi[3], Rudy Herteno[4], Setyo Wahyu Saputro[5]

[1,2,3,4,5] Department of Computer Science, Faculty of Mathematics and Natural Science, Lambung Mangkurat University, Banjarbaru, Indonesia

ARTICLE INFO

History of the article:
Received July 15, 2025
Revised August 20, 2025
Accepted January 27, 2026

Keywords:
API Security
Excessive Data Exposure
Fuzzing
Healthcare Application
OWASP Top 10

Correspondence:

E-mail:
radityo.adi@ulm.ac.id

ABSTRACT

Traditional perimeter security measures, such as Web Application Firewalls (WAFs) and static analysis, often fail to detect logic-based vulnerabilities in healthcare Application Programming Interfaces (APIs), creating significant risks for patient data confidentiality. Addressing the scarcity of empirical performance evaluations in this domain, this study employs a grey-box controlled experimental design to assess the effectiveness of automated HTTP fuzzing against a production-grade mobile health application ("Application X"). Using the FFUF tool configured with sequential identifier injection, status-code filtering, and hidden-field probing, the experiment tested 33 endpoints against the OWASP API Security Top 10 2023 benchmarks. To ensure data reliability, a rigorous multi-step validation protocol including replay testing and environmental noise elimination was applied to filter false positives. The results identified 88 distinct vulnerabilities distributed across six categories, with a critical dominance of Security Misconfiguration (API8) and Broken Object Property Level Authorization (API3). Analytically, the high prevalence of API3 reveals a systemic failure in backend serialization, where sensitive fields including password hashes and internal administrative flags were exposed due to the absence of Data Transfer Objects (DTOs), contradicting the assumption of secure client-side filtering. Limitations of this study include the restriction to a single patient-role perspective and the exclusion of third-party integrations. The study concludes that automated fuzzing is superior to static analysis in detecting runtime data leakage and recommends mandatory Server-Side Output Filtering through explicit DTOs as a critical standard for secure health API development and data privacy compliance.

1. INTRODUCTION

The rapid integration of mobile technology into the healthcare sector has positioned Application Programming Interfaces (APIs) as the critical backbone of digital health ecosystems. APIs facilitate essential interoperability between patient-facing mobile applications, hospital information systems, and third-party services. However, this architectural connectivity introduces significant security attack surfaces (Al-Rumaim & Pawar, 2024). The Open Web Application Security Project (OWASP) identifies that APIs are increasingly targeted due to vulnerabilities such as Broken Object Level Authorization and Security Misconfiguration, which can lead to catastrophic data breaches. Recognizing these distinct threats, the OWASP API Security Project has become the de facto standard for identifying risks that traditional web security models often overlook (Idris et al., 2022). In the context of healthcare, where data confidentiality is mandated by strict regulations, the security integrity of these interfaces is paramount.

Despite the critical nature of these systems, securing mobile health APIs remains a complex challenge. Traditional perimeter security measures, such as Web Application Firewalls (WAFs) and static code analysis, often fail to detect logic-based vulnerabilities that arise from improper implementation of business

rules or complex authorization flows. Recent studies emphasize that generating valid API call sequences is critical for uncovering these deep-state vulnerabilities, a capability often lacking in traditional scanners (Liu et al., 2022). Furthermore, manual penetration testing is frequently unscalable and inconsistent when addressing the vast number of endpoints in modern microservices architectures, which introduce distributed security challenges that monolithic testing approaches cannot adequately cover (Mateus-Coelho et al., 2021). A specific challenge observed in "Application X", a production-grade mobile health application utilized for clinical operations, is the reliance on intricate role-based access controls that are prone to implementation errors. Recent incidents, such as major healthcare data breaches in Indonesia, underscore the urgency of adopting more rigorous and automated testing methodologies. This aligns with global trends highlighting that current mobile defense mechanisms are often insufficient against evolving malware and data exfiltration tactics (Cinar & Kara, 2023).

Current research in API security has largely focused on general web applications or static analysis techniques. However, recent systematic reviews indicate that automated testing for authenticated RESTful APIs remains a significant challenge with limited coverage (Ehsan et al., 2022). Furthermore, the absence of standardized test coverage criteria for RESTful APIs hinders the objective assessment of these testing methodologies, making it difficult to compare vulnerability detection techniques effectively (Martin-Lopez et al., 2019). While fuzzing is a well-established technique for identifying memory corruption errors, its effectiveness in detecting logic and authorization flaws in RESTful APIs remains an area requiring deeper empirical investigation. Foundational studies suggest that stateful fuzzing, which infers producer-consumer dependencies between API requests, is essential for uncovering complex logic vulnerabilities that stateless scanners miss (Atlidakis et al., 2019). There is a distinct lack of controlled experimental data quantifying how well automated fuzzers can identify OWASP API Top 10 risks when applied to proprietary health applications that utilize complex authentication schemes.

To address this research gap, this study performs a grey-box controlled experiment to evaluate the effectiveness of automated HTTP fuzzing using the FFUF (Fuzz Faster U Fool) tool. Unlike exploratory case studies, this research applies a systematic configuration to detect specific vulnerability categories defined in the OWASP API Security Top 10 2023, including Broken Object Level Authorization (API1), Broken Authentication (API2), and Broken Object Property Level Authorization (API3), within the "Mobile Health Application X" environment. The use of automated fuzzing is proposed to overcome the limitations of manual testing by providing high-throughput and reproducible tests capable of uncovering edge-case vulnerabilities often missed by standard scans.

The specific objectives of this research are: (1) to measure the detection performance of automated fuzzing against the OWASP API Security Top 10 2023 benchmarks; (2) to analyze the severity and distribution of vulnerabilities found in a controlled sandbox environment; and (3) to provide empirical recommendations for hardening mobile health APIs against automated attacks. This study contributes to the body of knowledge by demonstrating the utility and limitations of fuzzing as a low-cost and high-impact security testing methodology for healthcare practitioners.

2. RESEARCH METHODS

The research methodology flow can be seen in Figure 1.

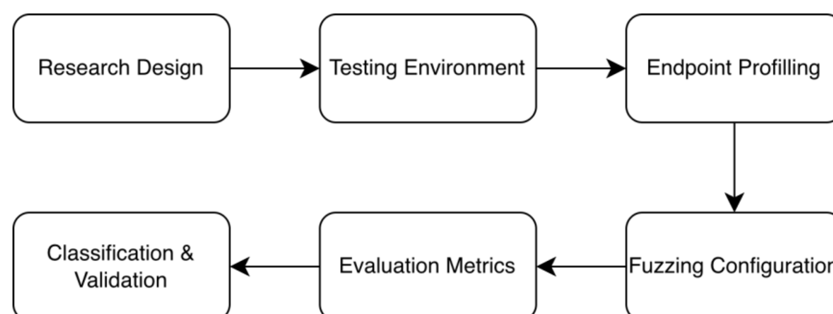

Figure 1. Research Methodology Flow

2.1. Research Design

This study employs a grey-box controlled experimental design to assess the effectiveness of automated HTTP fuzzing. This methodological choice is supported by recent systematic reviews indicating that dynamic analysis is essential for uncovering runtime behaviors and data leakage in Android ecosystems that static analysis often misses (Sutter et al., 2024). The researcher possessed partial internal knowledge specifically the API documentation, endpoint structures, and baseline payload schemas while having no

involvement in system development. This access level enables realistic evaluation without full white-box visibility.

All fuzzing executions were conducted under controlled experimental conditions, in which configurations, authentication states, network parameters, and fuzzing wordlists remained fixed across trials. The objective of the study is strictly performance evaluation of fuzzing for API vulnerability detection rather than organizational risk assessment. Authentication logic was preserved throughout testing to ensure that endpoints intended to be public were not misclassified as broken-authentication vulnerabilities.

2.2. Testing Environment

Experiments targeted Application X (API Version 1.14.0) deployed in a sandbox environment provided by the system operator. The system follows a conventional REST architecture backed by an ORM-managed relational database, served behind a reverse proxy. To preserve confidentiality, actual server infrastructure details and endpoint base paths are abstracted, while maintaining functional accuracy.

Fuzzing was executed on a macOS workstation (MacBook Pro 2021, 16 GB RAM, 512 GB SSD) using FFUF v2.1.0, over a stable broadband HTTPS connection.

Default server behaviors such as request throttling and timeout thresholds remained intact unless required for measurement stability. During testing, firewall-level automated blocking was temporarily disabled by the system operator to avoid distortion of throughput-sensitive measurements, whereas authentication, authorization, and token validation logic operated normally.

2.3. API Endpoint Profiling

Endpoint profiling was conducted exclusively using the official API documentation, where no hidden or administrative endpoints were enumerated and only those accessible through standard mobile application workflows were included. To prepare these endpoints for fuzzing, several normalization steps were applied in a continuous process. This began with reducing query parameters to their canonical minimal forms and removing pagination, ordering, or sorting parameters unless they were functionally required. Subsequently, JSON or multipart bodies were standardized to valid baseline payloads, and authentication headers were attached only when the documentation explicitly indicated protected access. All tests were further refined by executing them through a single authenticated patient-level account to ensure consistent privilege boundaries..

The identified endpoints were grouped into five technical categories, each mapped to specific fuzzing configurations and evaluation metrics as detailed in Table 1. While exact URLs are abstracted for confidentiality, the structures provided are representative of the actual system behavior.

Table 1. Endpoint categories, applied fuzzing configurations, and evaluation metrics

No	Category	Description	Fuzzing Configuration Applied	Evaluation Metrics
1	Standard retrieval endpoints	GET endpoints returning lists or single records with simple parameters.	Identifier/Object Access Fuzzing (API1); Authentication Enforcement Testing (API2).	Unauthorized Object Access Events (API1); Unexpected Access Grants (API2).
2	Relational retrieval endpoints	GET endpoints supporting nested relational expansion via filters.	Relational Query Fuzzing (API4).	Excessive Data Exposure Ratio (API4).
3	Authenticated mutation endpoints	POST/PATCH endpoints requiring valid tokens for creation or modification.	Hidden Property Injection Fuzzing (API3); Business Logic Flow Stressing d(API6).	Hidden Property Acceptance Frequency (API3); Business Logic Bypass Events (API6).
4	Workflow control endpoints	Endpoints responsible for multi-step processes or state transitions.	Business Logic Flow Stressing (API6).	Business Logic Bypass or Interruption Events (API6).
5	Error-prone parsing endpoints	Endpoints processing JSON, multipart, or complex payload structures.	Malicious & Invalid Payload Fuzzing (API8).	Error Message Disclosure Intensity (API8).

2.4. Fuzzing Configuration

The testing process began with Identifier and Object Access Fuzzing (API1: Broken Object Level Authorization), where sequential identifiers were injected into path parameters and ID-bearing fields. This was performed using the command ffuf -u https://target-api/resource/FUZZ -X GET -w ids.txt -mc 200-599 -t 20. This configuration was justified because sequential IDs effectively emulate common enumeration attacks, while the -t 20 parameter balanced throughput with server stability and the -mc 200–599 range ensured that both successful access and unexpected behaviors were captured. Following this, Authentication Enforcement Testing (API2: Broken Authentication) was conducted to determine if protected endpoints strictly enforced token checks using the command ffuf -u https://target-api/FUZZ -X GET -H "Authorization: <variant>" -w endpoint-list.txt -mc 200-599. Variants tested included no token, invalid tokens, and valid tokens, a strategy that remains critical as improper JWT signature verification and algorithm confusion are still prevalent vulnerabilities in modern RESTful architectures (Dalimunthe et al., 2023).

Simultaneously, Hidden Property Injection Fuzzing (API3: Broken Object Property Level Authorization) was performed by inserting undocumented JSON or multipart fields into mutation endpoints. This utilized the command ffuf -u https://target-api/entity/123 -X PATCH -H "Authorization: Bearer <token>" -d '{"baseline":"value","FUZZ":"injected"}' -w hidden-fields.txt -mc 200-599. The objective was to detect schema weaknesses, improper whitelisting, or the silent acceptance of unauthorized fields, adhering to principles of intelligent data fuzzing that emphasize generating semantically valid yet boundary-stressing payloads to bypass input validation layers (Godefroid et al., 2020). Furthermore, Relational Query Fuzzing (API4: Unrestricted Resource Consumption) applied invalid and deeply nested relations to stress relational expansion via the command ffuf -u "https://target-api/resource?filter={...FUZZ...}" -w relations.txt -mc 200-500. This test evaluated whether excessive server-side processing exposed performance risks or unintended data structures.

The final stages of the configuration included Business Logic Flow Stressing (API6: Unrestricted Access to Sensitive Business Flows) and Security Misconfiguration (API8). Workflow endpoints were fuzzed with out-of-order, repeated, or contradictory state values using the command ffuf -u https://target-api/workflow/step -X POST -H "Authorization: Bearer <token>" -d '{"state":"FUZZ"}' -w state-values.txt to identify improper enforcement of workflow sequencing. Finally, Malicious and Invalid Payload Fuzzing was executed to test malformed JSON, broken multipart boundaries, and edge-case strings using the command ffuf -u https://target-api/parser -X POST -d 'malformed FUZZ payload' -w special-strings.txt -mc 400-599. This exposure was designed to reveal error messages that might leak sensitive internal architectural details..

2.5. Evaluation Metrics

The evaluation process combines quantitative and qualitative indicators aligned with OWASP API Security Top 10 (2023). This approach adheres to recent frameworks for fuzzing assessment, which emphasize that standardized metrics and controlled experimental conditions are essential for comparing the efficacy of vulnerability detection tools (Eceiza et al., 2023). Metrics were applied consistently across fuzzing sessions to identify deviations in authorization, authentication, schema integrity, business-flow handling, or resource behavior.

Key metrics utilized in this study included the Object Access Integrity Metric, which determines whether manipulated identifiers return resources belonging only to the authenticated user and requires reproducible responses across repeated trials to confirm violations. The Authentication Bypass Validation Metric was used to compare responses between authenticated and unauthenticated access, where any endpoints returning 2xx or 3xx status codes without valid credentials were re-tested outside the fuzzing environment to confirm stability. Additionally, the Sensitive Data Exposure Metric utilized pattern-based analysis to identify unexpected or undocumented fields, such as personal identifiers or relational metadata, with a field considered exposed only when consistently reproduced. Performance and logic were further assessed through the Resource Saturation Metric, which monitored latency shifts, 500/504 errors, or throughput degradation during high-rate fuzzing, requiring consistency across multiple runs to classify resource-consumption weaknesses. The Workflow Misuse Detection Metric evaluated whether multi-step flows accepted repeated or structurally invalid transitions, requiring stable replication across all tested sequences. Finally, the Response Consistency Metric was applied to assess irregularities such as inconsistent error formatting, missing validation messages, or unexpected successful responses.

To minimize false positives, all preliminary findings were subjected to a brief validation protocol designed to confirm consistency and eliminate artifacts introduced by the testing environment. This protocol integrated replay testing to confirm reproducibility and noise elimination to rule out transient network or server fluctuations. Furthermore, schema cross-checking was employed to ensure that

unexpected fields, codes, or structures were genuine deviations from the documented or expected response format.

2.6. Finding Classification, Severity Assessment, and Validation

The classification framework involved mapping each confirmed anomaly to the most appropriate OWASP category based on the specific security property violated. These categories encompassed API1 for unauthorized object access, API2 for missing or inconsistent authentication enforcement, API3 for the modification or reflection of unauthorized fields, API4 for excessive data traversal or resource load, API6 for the misuse of business workflows, and API8 for security misconfiguration and verbose error exposure. Following classification, severity was determined using a simplified Low, Medium, and High scheme that aligns with OWASP's Likelihood × Impact framework and is widely utilized in academic security studies. Low severity was assigned to findings with limited technical impact or those difficult to reproduce, while Medium severity was reserved for reproducible deviations affecting integrity or controlled functionality. High severity was designated for vulnerabilities with an easily exploitable impact on confidentiality, integrity, or unauthorized access.

To prevent inflated reporting, duplicate consolidation rules were implemented where findings were consolidated only within the same endpoint when multiple payloads produced identical behavioral outcomes. Cross-endpoint consolidation was not applied because each endpoint constitutes a distinct attack surface. Subsequently, each candidate finding underwent a rigorous multi-step confirmation procedure aligned with the reproducibility and verification practices emphasized in the OWASP Testing Guide to ensure that only genuine, consistently reproducible vulnerabilities were reported. The confirmation process began with replay verification, where each anomalous request was repeated multiple times, both with and without concurrent load, to confirm stable behavior rather than incidental server conditions. This was followed by noise elimination to rule out transient network or server fluctuations by comparing responses across controlled network conditions and standardized baselines.

For endpoints protected by authentication, authentication and context validation were performed by testing the anomaly against valid, expired, or tampered tokens, as well as anonymous access, to confirm that the deviation was tied to input semantics rather than session state. Furthermore, schema and response consistency checks involved replaying the input with structural variations, such as modified nesting or encoding changes, to determine if the behavior correlated with specific parser weaknesses, validation bypasses, or unstable serialization paths. Only findings that passed all confirmation stages were deemed valid for inclusion in the final analysis, ensuring that the results were not artifacts of network instability, rate limits, or non-deterministic responses. This process was guided by specific validation indicators, including unexpected 2xx responses without authentication, the exposure of undocumented relational or personal data, latency spikes or intermittent 500/504 errors, and inconsistent or unstable response schemas. These indicators ensured a consistent and defensible interpretation of the overall fuzzing outcomes.

3. RESULTS AND DISCUSSION

3.1. Vulnerability Detection Performance and Distribution

The automated fuzzing campaign executed on 33 endpoints successfully identified a total of 88 unique vulnerability instances distributed across 6 out of the 10 OWASP API Security categories (2023). This outcome demonstrates that while the application maintains robust documentation standards, it remains susceptible to specific runtime data leakage and configuration vulnerabilities.

The detailed mapping of findings per endpoint is presented in Table 2. This mapping reveals that vulnerabilities are clustered primarily around data retrieval and parsing endpoints.

Table 2. Mapping of Findings Based on Endpoints

No	API	API1: 2023	API2: 2023	API3: 2023	API4: 2023	API5: 2023	API6: 2023	API7: 2023	API8: 2023	API9: 2023	API10: 2023
1	API1								X		
2	API2								X		
3	API3	X		X	X				X		
4	API4			X	X				X		
5	API5			X	X				X		
6	API6	X		X	X				X		
7	API7								X		

No	API	API1: 2023	API2: 2023	API3: 2023	API4: 2023	API5: 2023	API6: 2023	API7: 2023	API8: 2023	API9: 2023	API10: 2023
8	API8			X					X		
9	API9			X					X		
10	API10			X					X		
11	API11			X					X		
12	API12	X		X	X				X		
13	API13			X							
14	API14	X		X	X				X		
15	API15								X		
16	API16		X	X							
17	API17	X		X	X				X		
18	API18	X		X	X				X		
19	API1	X		X	X				X		
20	API20			X					X		
21	API21	X		X	X				X		
22	API22	X		X	X				X		
23	AP23	X		X	X				X		
24	API24				X						
25	API25				X						
26	API26				X						
27	API27	X	X	X	X				X		
28	API28						X		X		
29	API29	X	X	X	X						
30	API30	X	X	X	X						
31	API31	X		X	X				X		
32	API32	X		X					X		
33	API33										

As summarized in table 3, the highest density of vulnerabilities was observed in API8: Security Misconfiguration (28.4%) and API3: Broken Object Property Level Authorization (27.2%). The dominance of API8 is attributed to the default behavior of the development framework, which was configured with verbose error handling enabled. Fuzzing payloads containing malformed JSON structures consistently triggered stack trace leaks, revealing internal server paths and library versions. This aligns with recent security guidance which identifies verbose error messages as a critical reconnaissance vector, enabling attackers to map internal architectures and identify vulnerable dependencies before launching targeted exploits (Sconiers-Hasan, 2024).

Additionally, the campaign detected 15 instances of API1: Broken Object Level Authorization, which contribute to the broader distribution of identified risks across the tested endpoints as detailed in Table 3. These vulnerabilities allowed the fuzzing agent to access unauthorized patient records by sequentially iterating numerical identifiers in the API endpoint path. This pattern confirms that the application lacks server-side session validation for object access, relying solely on predictable client-side references. This finding is consistent with Putra et al. (2023), who demonstrated that failure to implement object-level permission checks on the backend is the primary cause of IDOR vulnerabilities, enabling attackers to harvest sensitive user data through automated enumeration

Table 3. Distribution of Findings by OWASP API 2023 Categories

No	OWASP TOP 10 API Security Risk - 2023	Total
1	API1:2023 - Broken Object Level Authorization	15
2	API2:2023 - Broken Authentication	4
3	API3:2023 - Broken Object Property Level Authorization	24

No	OWASP TOP 10 API Security Risk - 2023	Total
4	API4:2023 - Unrestricted Resource Consumption	19
5	API5:2023 - Broken Function Level Authorization	0
6	API6:2023 - Unrestricted Access to Sensitive Business Flows	1
7	API7:2023 - Server Side Request Forgery	0
8	API8:2023 - Security Misconfiguration	25
9	API9:2023 - Improper Inventory Management	0
10	API10:2023 - Unsafe Consumption of APIs	0
Total Findings		88

Meanwhile, the high prevalence of API3 (Broken Object Property Level Authorization) in this study specifically manifests as Excessive Data Exposure. Analysis reveals that the majority of API3 findings occurred on GET endpoints responsible for retrieving patient records. This indicates that the backend API returns complete database entities including internal administrative flags and authentication data by improperly relying on the mobile client to filter the data before presentation. The exposure of such sensitive PII aligns with historical findings by Papageorgiou et al. (2018) and is further corroborated by Tangari et al. (2021). In their large-scale analysis of medical applications, Tangari et al. confirmed that data leakage is a systemic issue in the global mobile health ecosystem, where a significant proportion of apps fail to implement adequate privacy controls against insecure API responses. This persistent vulnerability is further highlighted by Das & Camp (2025), whose recent analysis emphasizes that such insecure data practices remain pervasive in mobile healthcare, often exposing patient details to unauthorized observers.

Furthermore, significant vulnerabilities were identified in API4: Unrestricted Resource Consumption, with 19 documented instances. The absence of rate-limiting headers or throttling mechanisms on these endpoints presents a critical availability risk. As demonstrated in recent experimental studies on RESTful services, application-layer DDoS attacks can easily bypass traditional network firewalls by mimicking legitimate traffic patterns, making endpoint-level throttling a mandatory defense mechanism (Sivakumar & Thilagam, 2025).

Conversely, no vulnerabilities were detected in categories API5, API7, API9, and API10, a distribution that is attributable to the specific architectural characteristics and the defined scope of the experiment. Regarding API5 (Broken Function Level Authorization), the absence of findings resulted from the testing scope being strictly limited to endpoints accessible to the 'patient' role, as the fuzzing campaign did not attempt vertical privilege escalation against administrative endpoints outside the documented user scope. For API7 (Server-Side Request Forgery), no findings were recorded because the tested endpoints do not accept user-supplied Uniform Resource Identifiers (URIs) or external resource references for backend processing, which effectively neutralized the primary attack vector for SSRF. The fuzzing process also confirmed that all accessible endpoints corresponded accurately to the official documentation provided by the developers, meaning no undocumented "shadow" or "zombie" endpoints were discovered under API9 (Improper Inventory Management). Finally, the risks associated with API10 (Unsafe Consumption of APIs) were eliminated because the application architecture is self-contained and does not directly consume or proxy data from third-party upstream services, thereby preventing unsafe external data ingestion.

3.2. Severity Assessment and Technical Evidence

To evaluate the operational risk posed by these findings, each vulnerability was classified based on the severity criteria defined in the Research Methods (Likelihood × Impact). The assessment prioritizes the potential for confidentiality loss and unauthorized system control. Table 4 presents the severity distribution of the 88 confirmed findings.

Table 4. Severity Classification of Identified Vulnerabilities

No	Severity Level	Count	Dominant Categories	Impact Justification
1	High	19	API1, API2	Direct unauthorized access to patient records (Confidentiality Loss) or authentication bypass mechanisms.
2	Medium	44	API3, API4, API6	Exposure of sensitive internal metadata (Excessive Data Exposure) or resource saturation risks.
3	Low	25	API8	Exposure of technical error messages (Information Disclosure) without direct exploitation paths.

To illustrate the vulnerability mechanics, a representative finding for API3 is analyzed on a user profile retrieval endpoint (GET /api/v1/user/profile/{id}). While the mobile application interface is designed to display only non-sensitive profile information (e.g., name and email), the raw API response intercepted by the fuzzer revealed critical authentication data. This is evidenced by a typical fuzzed request: GET /api/v1/user/profile/1024 HTTP/1.1, directed at api.health-app-x.internal with a valid patient authorization token provided in the header.

The corresponding server response returned an HTTP/1.1 200 OK status, yet the JSON body exposed the user's full name and email alongside sensitive internal fields, including the hashed password and a null password reset token. Specifically, the response included: { "user_id": 1024, "full_name": "John Doe", "email": "john.doe@email.com", "role": "patient", "password": "$2y...[HASH_REDACTED]...", "password_reset_token": null, "created_at": "2024-01-15T08:30:00Z" }.

Root cause analysis indicates that the backend system serializes the complete User entity, including the hashed password field, directly to the HTTP response. This vulnerability persists because the ORM model lacks a "hidden" attribute configuration or a dedicated Data Transfer Object (DTO) to exclude sensitive columns from JSON serialization, resulting in an improper and insecure reliance on client-side filtering to hide sensitive data from the user interface.

3.3. Comparative Effectiveness

The results underscore the distinct advantage of automated fuzzing over traditional static analysis (SAST) in detecting runtime data leakage. Static analysis tools often fail to detect this vulnerability because they cannot accurately predict the runtime serialization behavior of database objects or the context in which the data is presented to the client. This finding corroborates recent studies demonstrating that black-box fuzzing approaches are significantly more effective and scalable in uncovering deep-state vulnerabilities in modern web applications compared to static methods (Alsaidi et al., 2022). However, this experiment demonstrates that fuzzing is superior in identifying Excessive Data Exposure (API3).

While the automated fuzzing approach demonstrated high efficiency in identifying exposure risks, it emphasizes the need for complementary verification. This aligns with findings from recent systematic reviews, which conclude that while automated vulnerability scanners are crucial for scalability, their effectiveness varies significantly across different vulnerability categories, necessitating a hybrid approach with manual validation (Alazmi & De Leon, 2022).

3.4. Implications for Secure Health API Development

The findings bear significant implications for healthcare application developers and data protection compliance. The prevalence of Excessive Data Exposure (API3) indicates a reliance on "security by obscurity" hiding data in the UI rather than implementing robust data filtering at the API level. To mitigate these risks, developers must prioritize several critical actions, starting with the implementation of server-side output filtering. Specifically, developers should employ Data Transfer Objects (DTOs) or response interceptors to explicitly whitelist output fields for every GET endpoint, ensuring that internal metadata and credentials are never transmitted to the client. Adopting such strict output filtering mechanisms is widely recognized as a critical best practice for maintaining the security and scalability of modern RESTful architectures (Gowda & Gowda, 2024).

Furthermore, production configurations must be hardened to suppress stack traces associated with API8 vulnerabilities, thereby preventing potential attackers from mapping the internal application architecture. Continuous inventory verification also remains essential; although API9 was not identified in this study, the ongoing maintenance of documentation as observed in the target application is necessary to prevent the emergence of shadow APIs in future iterations. Finally, while the fuzzing campaign successfully identified 88 vulnerabilities, the inherent limitations of automated dynamic analysis, particularly regarding

false positives, must be acknowledged. Consequently, future work should consider a hybrid approach integrating Static Application Security Testing (SAST) to cross-verify findings, as recommended by recent empirical studies to maximize vulnerability detection accuracy and reduce manual validation overhead (Feio & Pardal, 2024).

4. CONCLUSIONS AND RECOMMENDATIONS

This study empirically demonstrates that automated grey-box fuzzing is a highly effective methodology for identifying runtime logic and configuration vulnerabilities in mobile healthcare APIs. Of the 33 endpoints tested, the experiment successfully uncovered 88 vulnerabilities across six OWASP API Security 2023 categories. The findings were dominated by Security Misconfiguration (API8) and Excessive Data Exposure (API3), confirming that while the target system properly handles standard access controls, it critically fails to secure data serialization and error handling outputs.

This research advances the understanding of healthcare cybersecurity by providing evidence that automated fuzzing outperforms traditional static analysis in detecting "invisible" runtime flaws. Specifically, it highlights that logic-based vulnerabilities like Excessive Data Exposure (API3) where backend systems leak sensitive metadata or credentials cannot be identified by code scanning alone but are readily exposed through dynamic response analysis.

For developers and practitioners, the results imply that reliance on client-side filtering is a critical architectural flaw. To protect patient data compliance, it is imperative to implement Server-Side Output Filtering using Data Transfer Objects (DTOs) to explicitly whitelist public fields. Furthermore, production environments must enforce strict suppression of verbose error messages to prevent architectural mapping by attackers.

Beyond specific code fixes, the organization should transition from periodic vulnerability scanning to a continuous DevSecOps pipeline. Implementing a 'Shift Left' strategy, where automated fuzzing is integrated directly into the CI/CD build process, has been proven to significantly reduce remediation costs and exposure windows compared to traditional post-deployment testing (Manchana, 2024).

Future research should expand upon these findings by incorporating white-box testing with full source code access and multi-role fuzzing. Additionally, future iterations could leverage next-generation fuzzing frameworks like APIF, which extend beyond traditional REST constraints to offer more comprehensive vulnerability coverage (Wang & Xu, 2024).

Finally, to fundamentally harden the security posture against the vulnerabilities identified, future iterations of the platform should adopt a Zero Trust Architecture (ZTA). As proposed by Al-Naji et al. (2024), ZTA minimizes the impact of potential API breaches by enforcing continuous, context-aware verification for every request, ensuring that even if an endpoint is exposed, lateral movement within the healthcare network remains restricted."

REFERENCES

Al-Naji, M., Zagrouba, R., & Al-Otaibi, S. (2024). A zero trust architecture for health information systems. Health and Technology, 14, 189–199. https://doi.org/10.1007/s12553-023-00809-4

Al-Rumaim, A., & Pawar, J. D. (2024). Exploring the evolving landscape of API security challenges in the healthcare industry: A comprehensive review. IEEE Access, 12, 10456-10478. https://doi.org/10.1109/SIN60469.2023.10474998

Alazmi, S., & Leon, D. C. de. (2022). A Systematic Literature Review on the Characteristics and Effectiveness of Web Application Vulnerability Scanners. IEEE Access, 10, 33200–33219. https://doi.org/10.1109/ACCESS.2022.3161522

Alsaidi, A., Alhuzali, A., & Bamasag, O. (2022). Effective and scalable black-box fuzzing approach for modern web applications. Journal of King Saud University - Computer and Information Sciences. https://doi.org/10.1016/j.jksuci.2022.10.006

Atlidakis, V., Godefroid, P., & Polishchuk, M. (2019). RESTler: Stateful REST API fuzzing. In Proceedings of the 41st International Conference on Software Engineering (ICSE) (pp. 748-758). IEEE Press. https://doi.org/10.1109/ICSE.2019.00083

Augustine, N., Sultan, A. M., Osman, M., & Sharif, K. (2024). Application of artificial intelligence in detecting SQL injection attacks. International Journal on Informatics Visualization, 8(4), 2131–2138. https://doi.org/10.62527/joiv.8.4.3631

Cinar, A. C., & Kara, T. B. (2023). The current state and future of mobile security in the light of the recent mobile security threat reports. Multimedia Tools and Applications, 82, 20269–20281. https://doi.org/10.1007/s11042-023-14400-6

Dalimunthe, S., Putra, E. H., & Ridha, M. A. F. (2023). Restful API security using JSON Web Token (JWT) with HMAC-Sha512 algorithm in session management. IT Journal Research and Development, 8(1), 81–94. https://doi.org/10.25299/itjrd.2023.12029

Eceiza, M., Flores, J. L., & Iturbe, M. (2023). Improving fuzzing assessment methods through the analysis of metrics and experimental conditions. Computers & Security, 124, 102946. https://doi.org/10.1016/j.cose.2022.102946

Ehsan, A., Abuhaliqa, M. A. M. E., Catal, C., & Mishra, D. (2022). RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 12(9), 4369. https://doi.org/10.3390/app12094369

Feio, D., & Pardal, M. L. (2024). An empirical study of DevSecOps focused on continuous security testing. Proceedings of the 2024 IEEE International Workshop on Security and Trust Management (STM). https://doi.org/10.1109/EuroSPW61312.2024.00074

Godefroid, P., Huang, B.-Y., & Polishchuk, M. (2020). Intelligent REST API data fuzzing. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 725–736). ACM. https://doi.org/10.1145/3368089.3409719

Gowda, P., & Gowda, A. N. (2024). Best practices in REST API design for enhanced scalability and security. Journal of Artificial Intelligence, Machine Learning and Data Science, 2(1), 827–830. https://doi.org/10.51219/JAIMLD/priyanka-gowda/202

Idris, M., Syarif, I., & Winarno, I. (2022). Web application security education platform based on OWASP API security project. EMITTER International Journal of Engineering Technology, 10(2), 246–261. https://doi.org/10.24003/emitter.v10i2.705

Liu, Y., Li, Y., Deng, G., Liu, Y., Wan, R., Wu, R., et al. (2022). MOREST: Model-based RESTful API testing with execution feedback. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22) (pp. 1-12). ACM. https://doi.org/10.1145/3510003.3510133

Manchana, R. (2024). DevSecOps in cloud native cybersecurity: Shifting left for early security, securing right with continuous protection. International Journal of Science and Research (IJSR), 13(8), 1–8. https://www.researchgate.net/publication/383403159

Martin-Lopez, A., Segura, S., & Ruiz-Cortés, A. (2019). Test coverage criteria for RESTful web APIs. In Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '19) (pp. 15-21). ACM. https://doi.org/10.1145/3340433.3342822

Mateus-Coelho, N., Cruz-Cunha, M., & Ferreira, L. G. (2021). Security in microservices architectures. Procedia Computer Science, 181, 1225–1236. https://doi.org/10.1016/j.procs.2021.01.320

Papageorgiou, A., Strigkos, M., Politou, E., Alepis, E., Solanas, A., & Patsakis, C. (2018). Security and privacy analysis of mobile health applications: The alarming state of practice. IEEE Access, 6, 9390–9403. https://doi.org/10.1109/ACCESS.2018.2799522

Putra, R. A., Kautsar, I. A., Hindarto, H., & Sumarno, S. (2023). Detection and prevention of insecure direct object references (IDOR) in website-based applications. Procedia of Engineering and Life Science, 4, 1–7. https://doi.org/10.21070/pels.v4i0.1435

Sconiers-Hasan, M. (2024). Application programming interface (API) vulnerabilities and risks (Special Report CMU/SEI-2024-SR-004). Software Engineering Institute, Carnegie Mellon University. https://doi.org/10.1184/R1/25282342

Sivakumar, K., & Thilagam, P. S. (2025). Vulnerability testing of RESTful APIs against application layer DDoS attacks. International Journal of Advanced Computer Science and Applications, 16(3).

Sutter, T., Kehrer, T., Rennhard, M., Tellenbach, B., & Klein, J. (2024). Dynamic security analysis on Android: A systematic literature review. IEEE Access, 12, 57261–57287. https://doi.org/10.1109/ACCESS.2024.3390612

Tangari, G., Ikram, M., Sentana, I. W. B., Ijaz, K., Kaafar, M. A., & Berkovsky, S. (2021). Analyzing security issues of Android mobile health and medical applications. Journal of the American Medical Informatics Association, 28(10), 2074–2084. https://doi.org/10.1093/jamia/ocab131

Wang, Y., & Xu, Y. (2024). Beyond REST: Introducing APIF for comprehensive API vulnerability fuzzing. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '24). ACM. https://doi.org/10.1145/3678890.3678928