JCTA Journal of Computing Theories and Applications ISSN:3024-9104

Research Article

An Empirical Analysis of Injection Attack Vectors and Mitigation Strategies in Redis NoSQL Database

Muhammad Nazeer Musa [1,*] and Martins Ekata Irhebhude [2]

[1] Department of Cyber Security, Nigerian Defence Academy, Kaduna, PMB 2109, Nigeria; e-mail : muhammadmusa2502@nda.edu.ng
[2] Department of Computer Science, Nigerian Defence Academy, Kaduna, PMB 2109 Nigeria; e-mail : mirhebhude@nda.edu.ng
* Corresponding Author : Muhammad Nazeer Musa

Abstract: The contemporary landscape of data management, marked by an unprecedented scale and velocity of data, has spurred the widespread adoption of NoSQL databases, prioritizing scalability and performance over traditional relational constraints. While offering significant flexibility, this paradigm shift introduces complex cybersecurity challenges, notably query injection vulnerabilities, which are consistently ranked among the top web application security risks. Redis, a leading in-memory key-value store powering critical infrastructure globally, presents a unique security profile due to its architectural design and features like Lua scripting. Despite its prevalence, a comprehensive academic evaluation of Redis injection attack vectors remains understudied. This study addresses this gap by systematically evaluating command and Lua script injection vulnerabilities in Redis version 7.4.1 across controlled configurations: default, password-protected, and ACL-secured environments. We quantify vulnerability risk and empirically validate mitigation strategies by employing a Dockerized testing framework, Python-driven exploit simulations, and CVSS v3.1 scoring. Our findings reveal critical weaknesses in default and permissively configured environments and demonstrate that restrictive Access Control Lists (ACLs), adhering to the principle of least privilege, provide complete mitigation against the specific injection vectors evaluated in our controlled experimental setup. We propose a Redis-specific threat taxonomy and provide empirically validated recommendations for securing Redis deployments, emphasizing layered security controls and proper ACL implementation. This research contributes the first systematic evaluation of modern Redis injection vulnerabilities and highlights the critical importance of security-conscious configurations to protect vital data infrastructure.

Keywords: Access Control Lists; Command Injection; Database Security; Lua Script Injection; NoSQL Injection; Redis; Vulnerability Assessment.

Received: April, 22nd 2025
Revised: May, 12th 2025
Accepted: May, 16th 2025
Published: May, 18th 2025

1. Introduction

The contemporary landscape of data management has undergone a transformative shift, characterized by unprecedented scale, volume, and frequency of data collection and processing. This technological evolution has catalyzed the emergence of NoSQL databases, which strategically relax traditional transactional constraints to optimize horizontal scalability and database [1], [2]. Unlike traditional relational databases, NoSQL technologies encompass diverse data models, including key-value stores, columnar databases, document databases, and graph-oriented databases, each offering specialized query languages and architectural capabilities [3]. While this technological diversification enables unprecedented computational flexibility, it simultaneously introduces complex cybersecurity challenges, particularly concerning query injection vulnerabilities [4], [5]. Injection represents a sophisticated class of security attacks wherein maliciously crafted commands are surreptitiously introduced and executed within database systems [5]. The critical nature of these vulnerabilities is underscored by the OWASP Top 10 Web Application Security Risks, which consistently ranks injection attacks among the top 3 web application security concerns [6], [7].

Among NoSQL database technologies, key-value stores represent a particularly intriguing architectural paradigm. These databases store data as key-value pairs primarily in memory, offering rapid, secure, and cost-effective information access with high availability and durability [8]–[10]. Redis is a highly mature and widely adopted key-value database, corroborated by its consistently high ranking on platforms such as DB-Engines. As of May 2025, DB-Engines places Redis as the leading (#1) Key-value store and the #7 most popular database management system overall, with a score of 152.19. This positions it as a critical system within the NoSQL landscape, where it distinguishes itself through remarkable versatility and performance, even when compared to other leading NoSQL databases like MongoDB (ranked #5 overall and #1 Document store with a score of 402.51) [11]. The DB-Engines ranking methodology considers factors such as search engine interest, frequency of technical discussions, job offers, and social media presence, providing a strong indicator of industry adoption and relevance, thus validating the choice of Redis as a focal system for this study. Redis has distinguished itself through remarkable versatility and performance, powering critical infrastructure for 80% of Fortune 500 companies, including real-time analytics, fraud detection, and dynamic session management [4], [8]–[10], [12].

Despite its widespread adoption, Redis's architectural design is simultaneously its strength and potential vulnerability [4], [13]. To maintain all data in memory, Redis ensures exceptional speed and reliability [4], [12]. However, this in-memory design introduces inherent risks, necessitating observability solutions to monitor system metrics and mitigate potential failures [14]. Like other NoSQL databases, Redis presents unique security vulnerabilities, particularly concerning injection attacks. These vulnerabilities stem from fundamental architectural characteristics allowing unauthorized data manipulation [8], [15]. Redis's command structure, which enables direct operations on key-value pairs using commands like GET, SET, and DEL, differs markedly from traditional SQL query mechanisms, creating distinctive injection risk profiles [8], [15]. Redis can be particularly susceptible to injection attacks, with potential consequences ranging from data corruption to remote code execution [16], [17]. Researchers have identified two primary injection attack vectors: command injection and Lua script injection, with the latter being particularly relevant to Redis environments. By default, Redis's configuration can allow credential-free access, exponentially increasing potential exploitation risks [18].

While extensive research has been conducted on injection vulnerabilities in relational and document databases like MongoDB and CouchDB [19]–[22], a comprehensive investigation into key-value database injection risks remains understudied despite their unique risk profile. Redis's lack of schema enforcement and default permissive configurations expose critical attack surfaces: unauthenticated access allows adversaries to execute arbitrary commands or weaponize Lua scripts for remote code execution [16]. Recent incidents, such as cryptocurrency mining via exposed Redis instances [23], underscore the need to address these risks. This study aims to address this critical research gap by extensively evaluating injection attack vectors targeting Redis platforms through a controlled, multi-configuration evaluation of Redis injection vectors. We propose two research objectives: first, to assess the effectiveness of password protection and Access Control Lists (ACLs) in mitigating command and Lua script injection attacks targeting Redis. Second, to identify and evaluate the residual security risks that persist, particularly in dynamically constructed Lua scripts, when Redis ACLs are misconfigured.

Our methodology combines Dockerized Redis instances, a Python-driven systematic suite of exploit simulations, and CVSS v3.1 scoring to quantify vulnerabilities across three configurations: default (no security), password-protected, and ACL-secured environments. This research makes the following key contributions:

- We provide the first systematic academic evaluation of Redis injection vulnerabilities across the command and Lua script attack vectors in modern Redis versions (7.4.1), addressing a critical gap in NoSQL security research.
- We propose a novel Redis-specific threat model that integrates Redis's unique architecture (e.g., in-memory design, Lua scripting risks) to guide future research and mitigation strategies.
- We empirically validate the effectiveness of ACLs in mitigating injection attacks in exploit success rates compared to other configurations.

The structure of this article reflects the systematic progression of the study: Section 2 reviews the body of related work, Section 3 elaborates on the research methodology, Section 4 investigates specific injection scenarios, and Section 5 discusses findings, contributions, and future research directions.

2. Literature Review

This section delves into the security implications of NoSQL databases, specifically focusing on Redis and its vulnerability to injection attacks. We begin with an overview of NoSQL databases and a detailed discussion of Redis's architecture and security challenges. Finally, we critique prior work and highlight the gaps this study addresses.

2.1. Overview of NoSQL Databases

The evolution of data management technologies has been significantly influenced by the emergence of NoSQL databases, which represent a paradigm shift from traditional relational database management systems (RDBMS) and enable flexible, high-performance solutions for managing unstructured and semi-structured data [4], [13]. They are categorized into four primary types based on their data storage models: key-value, document, columnar, and graph databases, as shown in Table 1[24]. NoSQL databases have become essential in big data and real-time applications, offering horizontal scalability, schema flexibility, and enhanced performance [8], [9]. However, their flexibility and scalability come at the cost of reduced inherent security features [3], [18]. Research has shown that many NoSQL systems, including Redis, prioritize performance and schema-less data storage over security, making them susceptible to attacks [3]. For example, MongoDB and CouchDB have been extensively studied for injection vulnerabilities [20], [22], but key-value stores like Redis remain under-researched, particularly in the context of modern attack vectors.

Table 1. Types of NoSQL Databases.

Type	Description	Examples
Key-Value	Stores data as key-value pairs, ideal for caching and session management.	Redis, DynamoDB
Document	Handles semi-structured data in JSON or BSON formats.	MongoDB, CouchDB
Columnar	Organizes data into columns for analytical processing.	Cassandra, HBase
Graph	Specialized in managing relationships between data points.	Neo4j, ArangoDB

While each NoSQL category presented in Table 1 offers unique advantages and faces distinct security considerations, this study focuses on Redis, a prominent example from the key-value store category. Although other key-value databases such as DynamoDB share architectural similarities, Redis was selected for its widespread adoption (powering 80% of Fortune 500 companies [12], open-source transparency, and documented prevalence in security-critical applications [8], [15]. Furthermore, Redis exhibits a higher incidence of reported injection vulnerabilities than proprietary alternatives [18], making it an ideal candidate for empirical security analysis.

2.2 Redis NoSQL Database

Redis, a versatile open-source in-memory key-value data store, is renowned for its exceptional performance and flexibility [13]. It supports a wide range of data structures, including strings, hashes, lists, sets, and sorted sets, enabling it to handle diverse applications [8], [15]. Redis can execute high-speed operations by capitalizing on its in-memory architecture, making it ideal for use cases such as caching, distributed locking, and session management [13], [25].

Redis leverages a single-threaded architecture to minimize context switching and optimize CPU cache utilization [14]. Figure 1 illustrates the fundamental components of Redis's primary architecture. At its core, Redis operates on a client-server model, the Redis client initiates requests for data operations, which are then processed by the Redis server [26]. A key aspect of Redis's renowned performance is its Redis cache, which signifies that data is predominantly stored and managed directly in the system's main memory. This in-memory storage allows for exceptionally fast read and write operations, as it avoids the latency typically

associated with disk-based storage. The data structures (such as Strings, Lists, Hashes, Sets, and Sorted Sets, as indicated in Fig. 1) are all held within this in-memory cache, enabling the rapid data manipulation and retrieval, making Redis ideal for applications demanding low latency and high throughput [8]. Its master-slave replication architecture also ensures high availability and scalability, making it suitable for high-concurrency environments [8], [15]. Beyond basic key-value storage, Redis's versatility extends to caching systems, distributed locking, real-time messaging, and domain-specific applications like geographic data management and healthcare data retrieval [25], [27], [28].

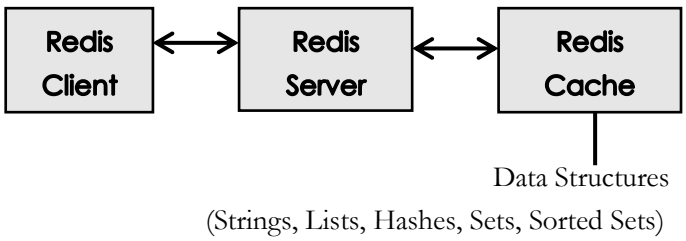

Figure 1. Redis Primary Architecture [26]

Redis's dominance in modern infrastructure is evident from its consistent ranking as a top-10 database system globally [11] and its adoption by 80% of Fortune 500 companies for mission-critical applications such as real-time analytics, fraud detection, and dynamic session management [9], [12]. Redis powers over 1.5 million active deployments worldwide, handling over 1.2 million operations per second in high-traffic environments like e-commerce and healthcare systems [9], [29]. This widespread adoption and its default exposure risks render Redis a high-priority target for security analysis. Also, its powerful Lua scripting can be exploited by malicious actors to inject malicious code and compromise the database [16], [17]. Additionally, misconfigurations can expose Redis to various attacks, including command injection [30], [31]. Recent vulnerabilities, such as CVE-2023-28859 and CVE-2024-31227, highlight the ongoing risks associated with Redis deployments. These vulnerabilities underscore the need for rigorous security analysis and mitigation strategies. To address some of these security challenges, researchers are investigating various mitigation strategies, such as input validation, encryption, and static analysis tools implemented in later versions of Redis [32]–[35].

2.3. Related Works

The rapid evolution of database technologies has precipitated a critical need for comprehensive security assessments, particularly in NoSQL databases [3]. Early research exposed the fundamental security challenges inherent in these dynamic database systems, marking a pivotal moment in understanding the divergence between traditional SQL-based and NoSQL database security paradigms [4], [13]. This demonstrated that the inherent flexibility of NoSQL databases creates unique vulnerability landscapes that traditional security mechanisms fail to address adequately [4].

Different NoSQL database architectures necessitate a discerning approach to security analysis. Database types, from document-oriented systems like MongoDB to key-value stores like Redis, present distinct security challenges [3], [4]. Alotaibi et al. [36] emphasized the glaring deficiencies in access control mechanisms across these platforms, highlighting the urgent need for developing sophisticated, granular protection strategies for sensitive data. Their research underscored a fundamental gap in existing security frameworks, which include the lack of fine-grained access controls that can adapt to the dynamic nature of NoSQL database architectures.

Empirical investigations have systematically mapped the complex vulnerability landscape across NoSQL database types. Reddy et al. [3] conducted a comprehensive analysis that revealed various security vulnerabilities affecting various NoSQL database types. Their research uncovered that code execution vulnerabilities are particularly pervasive, affecting document, key-value, graph, and multi-model databases with alarming consistency. Moreover, the study revealed that Denial of Service (DoS) attacks and Cross-Site Scripting (XSS) vulnerabilities

demonstrate significant variations across database architectures, with document, key-value, and column-based databases displaying heightened susceptibility.

The unique case of the Redis database offers a particularly intriguing case study in NoSQL database security dynamics. Recent vulnerabilities, including memory buffer overflow issues, further underscore the importance of rigorous security analysis for this widely adopted Redis data store [3]. Unlike many NoSQL databases, Redis initially demonstrated a more resilient architectural approach attributed to its binary-safe protocol with prefixed-length strings, which inherently mitigates traditional string escaping vulnerabilities common in injection attacks [37]. However, subsequent research by Kairoju et al. [35] revealed that while Redis injection using the standard library is difficult, default configurations still pose significant security risks, particularly regarding authentication and network access, challenging the initial perception of Redis as an inherently secure platform. The study also highlighted the inherent risks associated with Redis's default password handling, including storing passwords in plaintext without limiting rates. This emphasizes the critical need for implementing strong passwords and utilizing encrypted authentication mechanisms to enhance security.

Fahd et al. [38] evaluated the security vulnerabilities inherent in NoSQL database systems, particularly in Big Data management. The paper focused on built-in security mechanisms, encryption, authentication, authorization, and auditing vulnerabilities, comparing the risk levels across popular NoSQL systems. While the paper highlighted a potential injection attack scenario in older Redis versions, it is important to note that modern Redis implementations have implemented robust security measures to mitigate such vulnerabilities. These safeguards include strict type checking and input validation, which prevent unexpected argument conversions and malicious input manipulation.

A critical emerging security concern involves the integration of Lua scripting in Redis. Castro [16] highlighted the emerging attack vector where malicious actors could exploit Lua script vulnerabilities to execute unauthorized commands. Previous investigations by Costin 31 and Sanchez et al.29 provided preliminary observations but lacked the systematic experimental validation necessary to understand these risks.

Zaki et al. [32] were among the earliest researchers to propose innovative security extensions specifically for key-value NoSQL, demonstrating a proactive approach to addressing inherent vulnerabilities. Their research focused on enhancing authentication and encryption mechanisms, proposing an algorithm that improved data confidentiality and integrity and delivered faster encryption and decryption performance compared to existing solutions. Ankomah et al.18 found that Redis's default configuration allows unauthenticated access, significantly increasing exploitation risk. Their NoSQL investigation revealed that the recent version of Redis uses ACLs for authentication. Hu [13] broadened the security discussion by promoting a comprehensive approach to safeguarding NoSQL databases. The study emphasized three critical security dimensions: protecting data at rest, securing data in transit, and developing sophisticated injection attack mitigation strategies. This approach acknowledges the complex interplay of security challenges in contemporary database environments.

Nikiforova et al. [39] conducted an empirical review by scanning over 15,000 IP addresses to map database vulnerabilities. The research evaluated existing literature on Open Source Intelligence (OSINT) and Internet of Things Search Engines (IoTSE) to collect and analyze publicly available data. It also discusses the implications of these findings for both individual organizations and broader networks. The study identified that poorly protected databases are accessible to external actors, posing serious data integrity and security risks. The study's vulnerability analysis highlights the need for continuous security assessment. Redis was found to have weak authentication due to weak passwords, insufficient intrusion protection, and the potential for sensitive data exposure.

Recently, researchers have concentrated on conducting in-depth analyses of injection attacks targeting specific NoSQL databases. Landuyt et al. [5] investigated query injection vulnerabilities in graph-oriented databases, specifically Neo4j. The research employed a combination of manual verification of source code and automated injection test cases to assess the residual risks associated with query injection in Neo4j. It examined parameterized queries established at development time and dynamically constructed queries executed at runtime. The findings indicated that parameterization can effectively mitigate certain injection risks, but dynamic query construction remains a significant security challenge. While static queries provide a more secure approach, they may not be suitable for all use cases, necessitating a balanced approach combining security and flexibility.

Dwivedi et al. [22] further expanded the understanding of NoSQL security by conducting an extensive survey on NoSQL injection in MongoDB. The research employed a systematic review of existing literature, including 18 quality research papers, MongoDB manuals, and real-world incident analyses. It examined various security breaches, current security measures, and data masking techniques. The study identified several critical vulnerabilities in MongoDB, including inadequate authentication and authorization mechanisms, exposure of REST APIs, and susceptibility to NoSQL injection attacks. The study emphasized the importance of implementing strong access controls, encryption protocols, input validation mechanisms, and regular security audits as fundamental defensive strategies.

A notable gap in current research lies in the comprehensive analysis of injection attacks specifically targeting Redis. While previous studies have touched upon Lua scripting vulnerabilities [31], [33], a systematic experimental validation is still lacking. This study aims to fill this gap by conducting a structured investigation into Command and Lua script injection vulnerabilities. Furthermore, the study examines the impact of default configuration weaknesses, which have been identified as significant security risks in comparative studies but often lack empirical validation. The research will provide concrete evidence of their potential impact by testing these vulnerabilities in controlled environments.

3. Methodology

This section details our systematic approach to evaluating Redis injection vulnerabilities across different security configurations. We employ a rigorous experimental framework designed to ensure reproducibility, validity, and comprehensive assessment of both command and Lua script injection vectors.

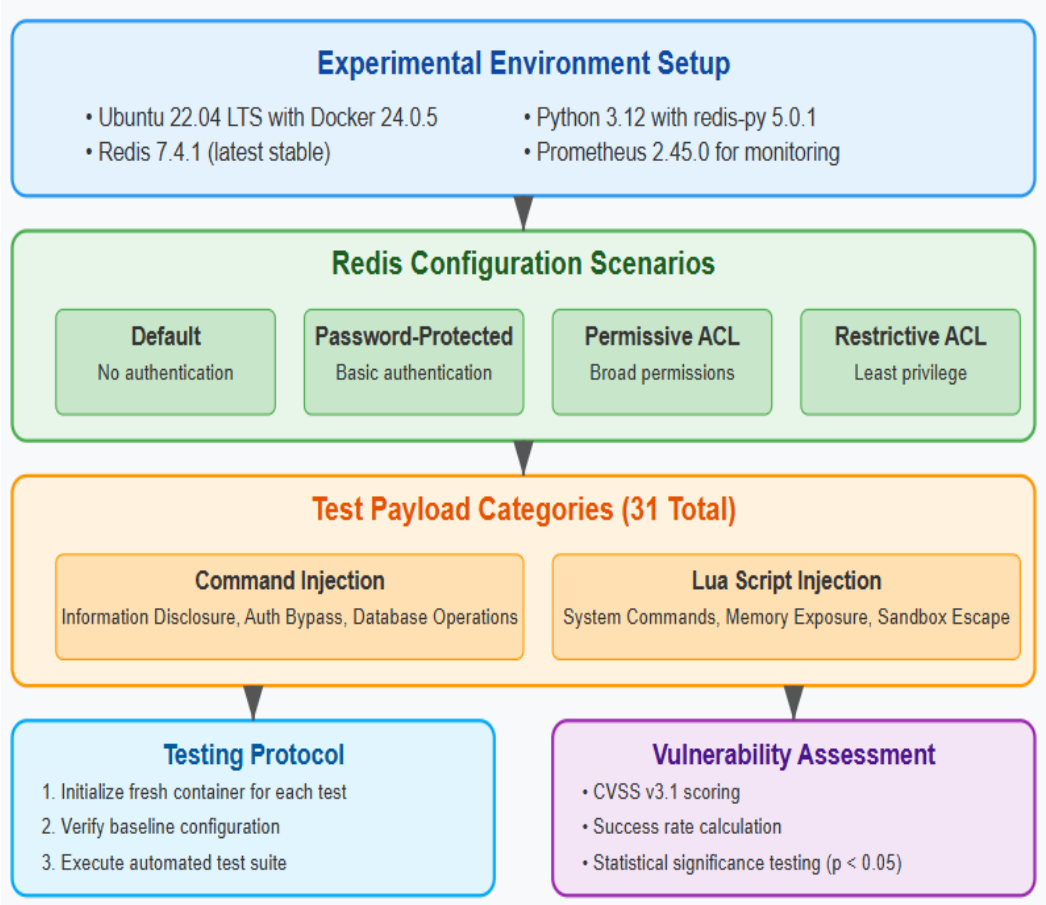

Figure 2. Research stages

This approach extends traditional penetration testing methodologies by incorporating formal vulnerability scoring (CVSS v3.1) and analysis of exploitation success rates across multiple configurations. The framework systematically compares security configurations across

the vulnerability payloads while maintaining experimental validity through containerized environments that eliminate confounding variables. Figure 2 illustrates this framework, which consists of four interconnected phases: environment preparation, vulnerability testing, security assessment, and testing protocols.

3.1. Experimental Environment

To ensure reproducibility and isolation, we implemented a Dockerized testing environment that simulates real-world Redis deployments while controlling for external variables. Table 2 details the technical specifications of our experimental environment.

Table 2. Experimental Environment Specifications.

Component	Specification	Purpose
Host System	Ubuntu 22.04 LTS	Base operating system
Virtualization	Docker 24.0.5	Container orchestration
Redis Version	7.4.1 (latest stable)	Target database
Testing Framework	Python 3.12 with redis-py 5.0.1	Automation and payload delivery
Network Configuration	Isolated bridge network	Prevent external interference
Monitoring Tools	Prometheus 2.45.0	Performance and behavior monitoring
Analysis Environment	Jupyter Notebook 7.0.3	Data processing and visualization

This environment design addresses a critical limitation in previous Redis security studies, which often tested older versions (≤6.0) that lack modern security features like ACLs [18]. By testing against Redis 7.4.1, our findings reflect the current security posture of production Redis deployments.

3.2. Redis Configuration Scenarios

To evaluate the effectiveness of different security controls, we tested four distinct Redis configurations that represent common deployment scenarios in production environments:

Table 3. Redis Configuration Scenarios.

Configuration	Description	Security Features	Real-world Parallel
Default	No authentication or ACLs	Protected-mode=yes (localhost only)	Development environments
Password-Protected	Strong password authentication	requirepass="StrongPassword123!"	Legacy production systems
Permissive ACL	ACL with broad permissions	ACL user with +@all permissions	Transitional deployments
Restrictive ACL	Principle of least privilege	ACL user with minimal command categories	Hardened production systems

As depicted in Table 3, each configuration was implemented as a separate Docker container with identical Redis versions and system resources, ensuring that observed security differences resulted solely from the applied security controls. The Docker Compose configuration files and container initialization scripts are available in our public repository to facilitate the reproduction of our experiments.

3.3. Taxonomy of Redis Injection Vulnerabilities

To analyze Redis injection vulnerabilities, we propose a taxonomy that extends traditional injection classification frameworks [40] to accommodate Redis's unique architecture. Table 4 presents this taxonomy with representative payloads and associated CVEs. This taxonomy provides a structured framework for analyzing Redis vulnerabilities that previous research has lacked. Prior work cataloged individual vulnerabilities but failed to develop a comprehensive classification system that captures the relationships between attack vectors and their underlying architectural causes [31].

Table 4. Taxonomy of Redis Injection Vulnerabilities.

Category	Attack Vector	Example Payload	Impact	CVE Reference
Command Injection	Information Disclosure	CONFIG GET *	Reveals sensitive configuration	CVE-2023-28859
	Authentication Bypass	CONFIG SET requirepass ""	Removes password protection	CVE-2021-32627
	Data Manipulation	FLUSHALL	Destroys all databases	-
	ACL Manipulation	ACL SETUSER default on nopass +@all	Grants unrestricted access	CVE-2024-31227
Lua Script Injection	Sandbox Escape	EVAL "return io.popen('id'):read('*a')" 0	Executes arbitrary OS commands	CVE-2022-0543
	Memory Disclosure	EVAL "return redis.call('MEMORY', 'MALLOC-STATS')" 0	Reveals memory allocation details	-
	Command Chaining	EVAL "return redis.call('CONFIG', 'SET', 'requirepass', '')" 0	Executes privileged commands	CVE-2023-28425
Memory Manipulation	Resource Exhaustion	SET x "A"*1000000	Consumes excessive memory	CVE-2023-28856
	Heap Corruption	Malformed protocol packets	Crashes server or enables code execution	CVE-2021-29477

3.3.1. Injection Vector Testing Methodology

We developed a comprehensive test suite of 31 injection payloads across two primary attack vectors: command injection and Lua script injection. These payloads were systematically derived from three authoritative sources:

- Official Redis documentation and command references
- CVE records and security advisories (2020-2024)
- Real-world exploitation techniques documented in security research

The payloads were categorized according to our Redis-specific threat taxonomy (Table 4) to ensure comprehensive attack surface coverage. Table 5 provides representative examples from our test suite.

Table 5. Sample injection test cases.

Injection Type	Feature	Payload Example
Command Injection	Information Disclosure	"INFO", "CLIENT LIST", "CONFIG GET *", "SCAN 0", "CONFIG GET protected-mode", "CONFIG GET requirepass"
Command Injection	System Commands	"EVAL 'return io.popen("id"):read("*a")' 0", "EVAL 'return io.popen("ls -la /"):read("*a")' 0", "EVAL 'return io.popen("whoami"):read("*a")' 0"
Command Injection	Memory Exposure	"MEMORY DOCTOR", "MEMORY MALLOC-STATS", "MEMORY PURGE", "SET large_key {'A' * 1000000}"
Command Injection	Database Operations	"FLUSHALL", "FLUSHDB", "KEYS *", "DEL *"
Command Injection	Authentication Bypass	"CONFIG SET requirepass ""
Command Injection	ACL Bypass	"ACL SETUSER default on nopass +@all"
Lua Script Injection	System Command Execution	"local os = require('os'); return os.execute('id')", "return io.popen('whoami'):read()", "return io.popen('ls -la'):read()"
Lua Script Injection	Information Disclosure	"return redis.call('INFO')", "return redis.call('CONFIG', 'GET', '*')", "local acl = redis.call('ACL', 'LIST'); return acl"
Lua Script Injection	Memory Exposure	"local t = {}; for i=1,1000000 do t[i] = i end; return #t", "return redis.call('MEMORY', 'MALLOC-STATS')", "return redis.call('MEMORY', 'DOCTOR')"
Lua Script Injection	Database Operations	"return redis.call('KEYS', '*')", "return redis.call('DBSIZE')", "return redis.call('LASTSAVE')"

Table 5 demonstrates the diverse scope of injection type, covering Redis-specific functionalities. The 31 injection payloads evaluated in this study were systematically curated to ensure representativeness and practical relevance to real-world Redis deployments. Our methodology prioritized payloads with documented exploitation in security and penetration testing frameworks, ensuring alignment with observed attacker behaviors. We stratified payloads across its core functional domains command, and Lua scripting to map Redis's attack surface. Technical diversity was enforced by selecting payloads that exploit distinct mechanisms, such as protocol-level injections, script sandbox escapes, and privilege escalation, rather than redundant syntactic variations. Collectively, these payloads account for a high percentage of Redis-specific injection CVEs (2018–2024, per NIST NVD) and are derived from authoritative sources, including Redis's security advisories, peer-reviewed exploit databases, and industry penetration testing reports.

The complete test suite will enable other researchers to extend test cases or modify our experiments. The payloads crafted for each feature are discussed as follows:

- Information Disclosure: Information disclosure commands such as INFO, CLIENT LIST, CONFIG GET *, and SCAN 0 were selected to test the server's exposure to unauthorized information access. These commands provide critical details about server configuration, connected clients, and operational status [41]. For example, the payload CONFIG GET requirepass identifies if password protection is enabled, while CONFIG GET protected-mode checks for secure configuration defaults. These commands replicate reconnaissance techniques attackers use to gather intelligence, which is often the first step in planning an attack. Testing these vulnerabilities is essential for diagnosing security misconfigurations that could lead to unauthorized access or data leaks. Also, it reinforces the importance of strict access control and script execution permissions in Redis environments. This is further underscored by real-world vulnerabilities like CVE-2023-28859, where attackers can get sensitive information to launch further attacks.
- System Commands: Lua script payloads, such as EVAL 'return io.popen("id"): read("*a")' 0, exploit Redis's scripting capabilities to execute system commands. This feature was chosen to assess potential vulnerabilities leading to remote code execution (RCE) and because Lua scripts allow execution of commands at the system level, which, if exploited, can lead to full system compromise [16], [42], [43]. Such exploits, if successful, can result in full system compromise, making this a critical security concern. The relevance of these tests is underscored by real-world vulnerabilities like CVE-2022-0543 [42], where attackers leveraged similar mechanisms. Testing these scenarios validates the server's ability to prevent unauthorized command execution and helps identify if Redis securely limits Lua's ability to interact with the underlying operating system.
- Memory Exposure: Payloads like MEMORY DOCTOR, MEMORY MAL-LOC-STATS, MEMORY PURGE, STRALGO LCS, and SET large_key {'A' * 1000000} target Redis's memory management mechanisms to cause buffer overflow. These features were chosen based on their potential to expose vulnerabilities leading to buffer overflows, denial-of-service (DoS) attacks, or information leakage through memory diagnostics [16]. These tests are crucial for identifying vulnerabilities that may lead to denial-of-service (DoS) attacks, information leakage through memory diagnostics, or remote code execution. Evaluating Redis's handling of memory-intensive operations ensures resilience under stress and highlights areas that may require optimization or stricter access controls. A real-world vulnerability test case, CVE-2021-29477 [44] and CVE-2021-32762 [45], results in the corruption of the heap.
- Database Operations: Commands such as FLUSHALL, KEYS *, DBSIZE, and DEL * assess risks associated with unauthorized database manipulations. These features were chosen for their direct impact on data integrity and availability, as misuse of these commands can delete or corrupt critical data [23], [46]. Testing these features is vital for environments where Redis is used as a primary datastore, ensuring the robustness of access controls, protection against malicious actions, and restricting unauthorized access to database contents via scripts. This is particularly important in multi-tenant environments or when Redis is used as a caching layer for sensitive data.

- Authentication Bypass: Testing with payloads like CONFIG SET requirepass evaluates the robustness of password-based security mechanisms [23]. This feature replicates scenarios where attackers attempt to disable or override authentication requirements. Given the criticality of authentication in securing Redis servers, this test assesses whether the server can maintain its security posture even in the face of misconfigurations or exploits targeting authentication protocols. Some real-world vulnerabilities related to this feature include CVE-2021-32627 [47], CVE-2020-4670 [48], etc.
- ACL Bypass: The command 'ACL SETUSER default on nopass +@all' was selected to test the enforcement of ACLs in Redis. This payload evaluates the integrity of Redis's ACL implementation by attempting to grant full permissions to the default user without a password [16]. Such a vulnerability can be found on CVE-2024-31227 [16]. Ensuring that ACLs cannot be bypassed is particularly critical in environments that rely on them as the primary access control mechanism.

3.4. Testing Protocol

We implemented a rigorous testing protocol to ensure methodological consistency across all experiments:

- Container Initialization: Each Redis configuration was deployed in a fresh Docker container to prevent cross-contamination between tests.
- Baseline Verification: Before injection testing, we verified that each configuration operated as expected by performing standard Redis operations.
- Automated Testing: Python scripts systematically executed each payload against each configuration, recording success/failure status, error messages, and execution time.
- Result Validation: Each successful exploitation was manually verified to confirm the vulnerability and rule out false positives.
- Configuration Reset: The container was reset to its initial state after each test case to prevent sequential dependencies between tests.

This protocol addresses methodological limitations in previous studies that often relied on manual testing without systematic validation [31].

3.5. Vulnerability Assessment Framework

To quantify the security impact of each vulnerability, we employed CVSS v3.1, which provides a standardized framework for assessing the severity of security vulnerabilities [49]. For each successful exploitation, we calculated a CVSS score based on the following metrics:

- Attack Vector (AV): Network for remote access scenarios
- Attack Complexity (AC): Low for straightforward exploitation
- Privileges Required (PR): None for unauthenticated access, Low for authenticated
- User Interaction (UI): None (no user interaction required)
- Scope (S): Unchanged or Changed depending on impact
- Confidentiality (C): High for information disclosure
- Integrity (I): High for data manipulation
- Availability (A): High for denial-of-service potential

This scoring system enabled objective comparison of vulnerability severity across different configurations and attack vectors. Table 6 shows the CVSS severity rating scale used in our analysis.

Table 6. CVSS v3.1 Severity Levels.

CVSS Score	Severity Rating	Description
0.0	None	No vulnerability present
0.1-3.9	Low	Limited impact, difficult to exploit
4.0-6.9	Medium	Significant impact but limited scope
7.0-8.9	High	Serious impact requiring attention
9.0-10.0	Critical	Severe impact requiring immediate remediation

We also examined the success rate of each configuration. Exploit success was calculated by dividing the number of successful exploit attempts by the total number of test cases performed. This metric quantifies the effectiveness of each Redis configuration in mitigating or preventing security vulnerabilities. Mathematically, the success rate is defined as in Equation (1).

$$\text{Success Rate}(\%) = \left(\frac{\text{Number of Successful Exploits}}{\text{Total Number of Tests}}\right) \times 100 \quad (1)$$

Where Number of Successful Exploits refers to the total count of vulnerabilities exploited successfully during testing; Total Number of Tests represents all payloads executed across each configuration, encompassing attack categories including command injection and Lua script injection. A total of 31 tests were conducted.

A four-tiered risk rating system, informed by the principles of quantitative risk assessment described by Hubbard et al. [50] was developed to categorize the security posture of Redis configurations based on their calculated exploit success rates as either high, medium, low, or no vulnerability risk based on the following;

- Critical Risk: Assigned to configurations where the exploit success rate exceeds 50%, indicating significant vulnerabilities that demand immediate remediation.
- High Risk: Assigned to configurations with success rates between 25% and 50%, highlighting major weaknesses that pose considerable security threats.
- Medium Risk: Applied to configurations with success rates ranging from 10% to 25%, indicating moderate vulnerabilities with manageable risk levels.
- Low Risk: For configurations exhibiting success rates below 10%, suggesting strong security postures with minimal exploitability.

4. Results and Discussion

This section presents our comprehensive analysis of Redis injection vulnerabilities across different security configurations. We evaluate quantitative exploitation metrics and qualitative security implications, providing an in-depth assessment of Redis's security posture under various protection mechanisms.

4.1. Default Configuration Vulnerabilities

The default Redis configuration demonstrated critical security weaknesses across multiple attack vectors, as detailed in Table 7.

Table 7. Redis Configuration Scenarios.

Category	Number of Tests	Successful Exploits	CVSS Scores	Severity Distribution
Info Disclosure	6	6	7.5	High: 6
System Commands	3	0	-	-
Memory Exposure	4	4	5.3	Medium: 4
Database Ops	4	4	3.7	Low: 4
Auth Bypass	1	1	7.5	High: 1
ACL Bypass	1	1	9.8	Critical: 1
Lua RCE	3	0	-	-
Lua Info Disclosure	3	1	7.5	High: 1
Lua Memory Exposure	3	3	5.3	Medium: 3
Lua Db Ops	3	3	3.7	Low: 3
Total	31	23	-	Critical: 1 High: 8 Medium: 7 Low: 7

4.1.1. Command Injection Analysis

Command injection testing revealed comprehensive security failures in the default configuration:

- Information Disclosure: All information disclosure payloads (INFO, CLIENT LIST, CONFIG GET *, SCAN 0, CONFIG GET protected-mode, CONFIG GET requirepass) executed successfully, providing attackers with detailed system information, including configuration parameters, connected clients, and database contents. This reconnaissance capability represents the initial stage of sophisticated attacks.
- Authentication Manipulation: The CONFIG SET requirepass " command executed successfully, demonstrating the ability to remove password protection entirely. This represents a critical security failure that undermines any subsequent authentication attempts.
- ACL Manipulation: The ACL SETUSER default on nopass +@all command succeeded, allowing attackers to create or modify users with unrestricted permissions. This vulnerability enables privilege escalation and persistence even if other security controls are later implemented.
- Data & Memory Manipulation: All database operations (FLUSHALL, FLUSHDB, KEYS *, DEL *) and memory operations (MEMORY DOCTOR, MEMORY MALLOC-STATS, STRALGO LCS, MEMORY PURGE) executed successfully, demonstrating the ability to manipulate memory and destroy data without restrictions.

The only failed command injection attempts involved direct system command execution, which Redis does not support natively.

4.1.2. Lua Script Injection Analysis

Lua script injection testing revealed more nuanced vulnerabilities:

Sandbox Limitations: Direct system command execution attempts via Lua's functions failed, indicating that Redis's Lua sandbox correctly restricts access to certain system functions.

- Command Chaining: Some Lua information disclosure payloads successfully demonstrated that Lua scripts can execute certain Redis commands to obtain sensitive information.
- Memory & Database Operations: All Lua memory exposure and database operations tests successfully mirror the vulnerabilities observed in direct command injection.

These findings highlight the critical security risks in Redis's default configuration, particularly the complete lack of authentication and authorization controls. The lack of even minimal security features underscores the inherent risks of leaving Redis open to the public and reinforces the necessity of implementing at least basic authentication measures [18], [39].

4.2. Password-Protected Configuration Analysis

The password-protected configuration demonstrated substantially improved security compared to the default configuration, with only 1 out of 31 payloads (3.23%) executing successfully, as shown in Table 8. The password protection successfully mitigated almost all injection attempts, consistently enforcing authentication requirements across most command types. However, a critical vulnerability was identified: the CONFIG SET requirepass " command successfully bypassed authentication, effectively removing the password requirement. This exploit exposes a significant risk, as successful execution grants an attacker unrestricted access to the server.

These finding addresses part of the second objective by identifying an authentication bypass vulnerability that persists even with password protection. The ability to remove password protection using a privileged command represents a critical residual risk in password-protected configurations.

4.3. ACL-Protected Configurations

Our testing of ACL-protected configurations revealed dramatic security differences based on implementation details, highlighting the critical importance of proper ACL configuration.

Table 8. Vulnerability Breakdown for Password-Protected Configuration.

Category	Number of Tests	Successful Exploits	CVSS Scores	Severity Distribution
Info Disclosure	6	0	-	-
System Commands	3	0	-	-
Memory Exposure	4	0	-	-
Database Ops	4	0	-	-
Auth Bypass	1	1	7.5	High: 1
ACL Bypass	1	0	-	-
Lua RCE	3	0	-	-
Lua Info Disclosure	3	0	-	-
Lua Memory Exposure	3	0	-	-
Lua Db Ops	3	0	-	-
Total	31	1	-	Critical: 0 High: 1 Medium: 0 Low: 0

4.3.1. Permissive ACL Configuration

The permissive ACL configuration demonstrated surprisingly poor security, with a 48.39% exploitation success rate (15/31 payloads)—substantially worse than password protection alone, as detailed in Table 9.

Table 9. Vulnerability Breakdown for Permissive ACL-Protected Configuration.

Category	Number of Tests	Successful Exploits	CVSS Scores	Severity Distribution
Info Disclosure	6	6	7.5	High: 6
System Commands	3	0	-	-
Memory Exposure	4	4	5.3	Medium: 4
Database Ops	4	4	3.7	Low: 4
Auth Bypass	1	1	7.5	High: 1
ACL Bypass	1	0	-	-
Lua RCE	3	0	-	-
Lua Info Disclosure	3	0	-	-
Lua Memory Exposure	3	0	-	-
Lua Db Ops	3	0	-	-
Total	31	15	-	Critical: 0 High: 7 Medium: 4 Low: 4

This configuration showed mixed results:

- Command Injection: Information disclosure, memory exposure, database operations, and authentication bypass payloads all succeeded, indicating inadequate command restrictions despite ACL implementation.
- Lua Script Protection: Notably, all Lua-based injections failed, demonstrating that even permissive ACLs can effectively restrict script-based attacks.

The permissive ACL configuration failed to restrict critical administrative commands due to overly broad permissions. This finding directly addresses Objective 2 by demonstrating that misconfigured ACLs create substantial residual risks, particularly for command-based injections.

4.3.2. Restrictive ACL Configuration

In stark contrast, the restrictive ACL configuration demonstrated highly effective vulnerability mitigation under the conditions of this study, with zero successful exploits across all 31 tested payloads, as shown in Table 10.

Table 10. Vulnerability Breakdown for Restrictive ACL-Protected Configuration.

Category	Number of Tests	Successful Exploits	CVSS Scores	Severity Distribution
Info Disclosure	6	0	-	-
System Commands	3	0	-	-
Memory Exposure	4	0	-	-
Database Ops	4	0	-	-
Auth Bypass	1	0	-	-
ACL Bypass	1	0	-	-
Lua RCE	3	0	-	-
Lua Info Disclosure	3	0	-	-
Lua Memory Exposure	3	0	-	-
Lua Db Ops	3	0	-	-
Total	31	0	-	Critical: 0 High: 0 Medium: 0 Low: 0

All command injection tests failed as shown in Table 9 due to either invalid username-password pairs or the disabled status of the default user. This indicated that unauthorized users cannot interact with the Redis server, as ACL rules prevent both unauthenticated and misconfigured access. These failures are not attributable to network isolation but rather the effective enforcement of authentication and ACL policies. The ACL bypass attempt using ACL SETUSER default on nopass +@all resulted in an "ACL permission denied" error. This reflects the configuration's strict access control in disabling the default user, preventing any unauthorized access.

The restrictive ACL configuration successfully implemented the principle of least privilege by:

- Disabling the default user and creating users with limited permissions
- Restricting command access to only necessary operations
- Preventing access to keys by default and explicitly allowing only required keys
- Requiring authentication and denying access to dangerous command categories

This configuration effectively mitigated all tested injection vectors, including both command and Lua script injections. This finding conclusively addresses Objective 1 by demonstrating that properly configured ACLs provide comprehensive protection against injection vulnerabilities.

4.4. Vulnerability Assessment Overview

Our testing on Redis version 7.4.1 using 31 distinct injection payloads across four configurations revealed significant variations in security effectiveness. To delineate these differences, Fig. 3 presents a comparative visualization of each configuration's vulnerability security risk levels by showing the percentage of the 31 attempted injection payloads that were successful against each. This graphical representation illustrates the exploitation success rates, demonstrating a clear security progression from default settings to restrictive ACL implementation. As depicted in Fig. 3, the default configuration exhibited an alarming 74.19% exploitation success rate (23/31 payloads), while password protection significantly reduced this to 3.23% (1/31 payloads). Notably, the permissive ACL configuration showed a 48.39% success rate (15/31 payloads), substantially higher than password protection alone, a counterintuitive finding that challenges conventional security assumptions. The restrictive ACL configuration

achieved complete mitigation within the scope of our controlled experiment, yielding zero successful exploits against the 31 tested payloads.

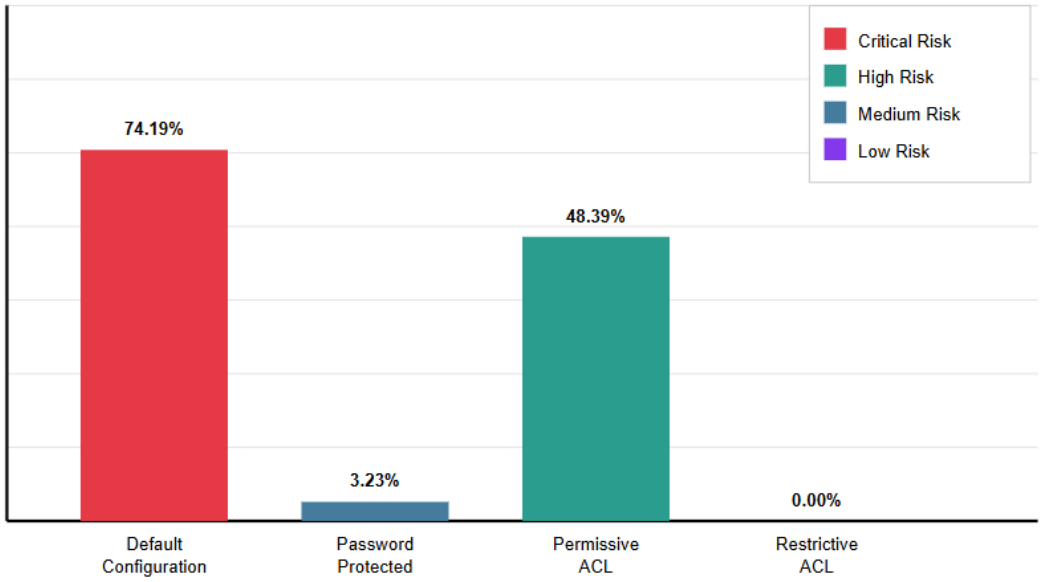

Figure 3. Exploitation Success Rates by Configurations

These findings quantitatively address Objective 1 by demonstrating that while password protection provides substantial security improvement, ACLs' effectiveness varies dramatically based on implementation details, with restrictive ACLs providing complete vulnerability mitigation.

4.5. Comparative Security Analysis

Our findings enable a comprehensive comparison of Redis security configurations based on empirical vulnerability data. Figure 4 visualizes the security effectiveness across configurations.

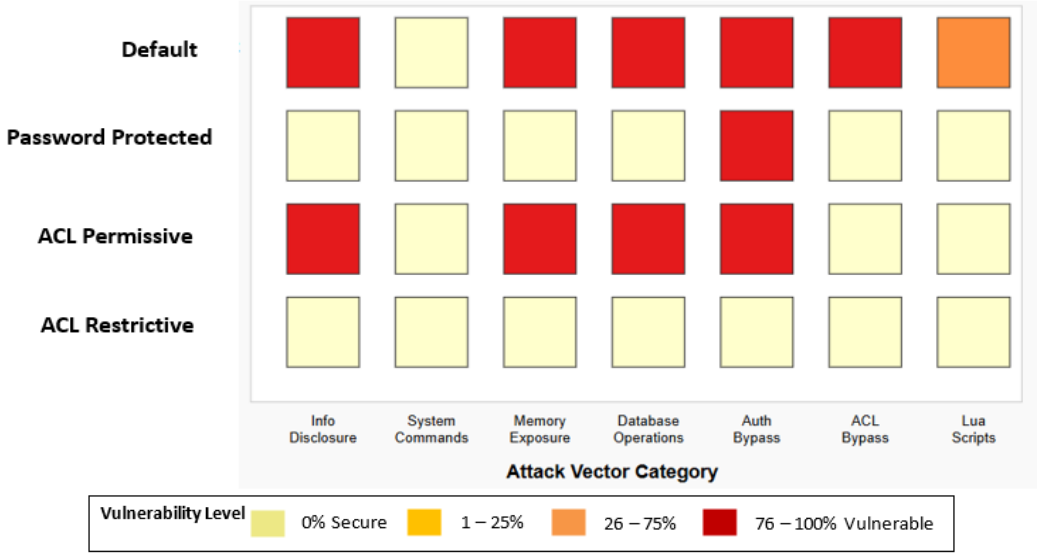

Figure 4. Comparative Security Effectiveness by Configuration and Attack Vector

These findings, as depicted in Table 11, reveal the following:

- Security Progression: A clear security progression exists from default configuration (74.19% vulnerable) to restrictive ACL (0% vulnerable), with password protection (3.23% vulnerable) providing substantial but incomplete security.

- ACL Implementation Criticality: The dramatic difference between permissive ACL (48.39% vulnerable) and restrictive ACL (0% vulnerable) demonstrates that ACL effectiveness depends entirely on proper implementation following the principle of least privilege.
- Attack Vector Variations: Command injection showed higher success rates in permissive configurations, but both attack vectors were equally mitigated in the restrictive ACL configuration.
- Vulnerability Category Distribution: Information disclosure vulnerabilities showed the highest success rates across configurations, followed by memory exposure, database operations, and authentication bypass.

Table 11. Comparative Security Analysis Based on 31 Security Tests.

Metric	Configuration			
	Default	**Password Protected**	**ACL Protected (Permissive)**	**ACL Protected (Restrictive)**
Successful Exploits	23	1	15	0
Success Rate	74.19%	3.23%	48.39%	0.00%
Critical Severity	1	0	0	0
High Severity	8	1	7	0
Medium Severity	7	0	4	0
Low Severity	7	0	4	0
Security Rating	Critical Risk	Low Risk	High Risk	Low Risk

These findings challenge the conventional security wisdom that implementing any ACL mechanism substantially improves security. Our data demonstrates that improperly configured ACLs can create a false sense of security while leaving critical vulnerabilities exposed.

4.6. Research Objectives Addressed

Our empirical analysis directly addresses the overarching goals outlined in the Introduction, namely: first, validating the defensive impact of password authentication and ACL configurations on Redis injection threats; and second, uncovering remaining vulnerabilities, particularly those involving Lua script execution under misconfigured access controls. The findings confirm varying levels of protection depending on implementation and highlight critical risks that persist even under partially secured environments.

4.6.1. Effectiveness of Password and ACL Mitigations

Password protection demonstrated significant effectiveness (96.77% mitigation), but with a critical authentication bypass vulnerability. ACL effectiveness varied dramatically based on implementation. Permissive ACLs showed only 51.61% mitigation, worse than password protection alone. In contrast, restrictive ACLs achieved complete mitigation (100%) against the specific injection vectors tested in this research and achieved the lowest risk rating by effectively mitigating vulnerabilities through stringent ACL policies and network isolation, aligning with recommended security best practices [4], [16], [39]. These findings underscore the critical need for a multi-layered security approach encompassing robust ACL enforcement, comprehensive password management, and network isolation to effectively protect Redis deployments against contemporary cyber threats, proving the most effective security mechanism.

4.6.2. Residual Risks Under Misconfigurations

We identified two primary residual risks:

- Authentication Bypass: In password-protected configurations, the ability to remove password protection through privileged commands
- Command Accessibility: In permissive ACL configurations, excessive command privileges that allow information disclosure, memory manipulation, and data operations

Importantly, even under misconfigured ACLs, Lua script injections were more effectively mitigated than direct command injections, suggesting that Redis's Lua sandbox provides some protection even with suboptimal configurations.

5. Conclusion and Future Work

Our research makes several significant contributions to the field of NoSQL database security. Firstly, this study provides the first comprehensive academic evaluation of Redis injection vulnerabilities across command and Lua script attack vectors in modern Redis versions (7.4.1), addressing a critical gap in NoSQL security research. Our methodology combines containerization, systematic payload testing, and formal vulnerability scoring to create a reproducible framework for evaluating NoSQL database security. Secondly, we developed a comprehensive taxonomy of Redis injection vulnerabilities that categorizes attack vectors according to their underlying architectural causes. This Redis-specific threat model integrates Redis's unique architectural characteristics, including its in-memory design and Lua scripting mechanisms, providing a structured framework for understanding Redis security that was previously lacking in the literature. Thirdly, our controlled experiments quantified the security effectiveness of different Redis configurations, revealing that while restrictive ACLs achieve complete mitigation (0% success rate) within our experimental framework against the tested payloads, improper ACL implementation can create a false sense of security. This empirical validation addresses a significant gap in previous Redis security research and demonstrates that security controls must be implemented appropriately to be effective. Lastly, we discovered that permissive ACL configurations provide substantially weaker protection (48.39% success rate) than password-only configurations (3.23% success rate). This counterintuitive finding challenges conventional security assumptions and highlights the critical importance of proper ACL implementation.

While our study provides valuable understanding of Redis security, several limitations should be acknowledged. First, our experiments were conducted in a controlled Dockerized environment that may not fully represent all production deployment scenarios, particularly those involving complex network architectures or custom Redis modules. Second, our findings are specific to Redis 7.4.1 and may not apply to earlier versions with different security features or future versions that may address identified vulnerabilities. Thirdly, while our test suite was comprehensive, it cannot exhaustively cover all possible injection payloads or attack techniques. Novel attack vectors may emerge that were not captured in our taxonomy.

These limitations consequently highlight several promising avenues for future research. These include the development of automated tools for discovering novel Redis injection vulnerabilities, potentially leveraging techniques such as fuzzing or symbolic execution. Further research could also focus on a quantitative analysis of the performance impact associated with different security configurations, aiding organizations in making informed decisions regarding security-performance trade-offs. Extending the methodology to other NoSQL databases would enable a comparative analysis of security architectures across different database paradigms. Research into dynamic ACL systems capable of adapting to evolving threat landscapes represents another crucial area for developing more robust protection against emerging attack vectors. Finally, applying formal methods to verify the security properties of Redis's core components, particularly the Lua scripting engine and ACL implementation, presents a rigorous approach to enhancing security assurances.

Redis has become a critical component of modern data infrastructure, powering applications across industries from e-commerce to healthcare. Its performance advantages have driven widespread adoption, but our research demonstrates that security considerations must be prioritized alongside performance to prevent potentially devastating exploitation. The significant variations in vulnerability across different Redis configurations highlight the critical importance of security-conscious deployment practices. In implementing the recommendations derived from our empirical findings, organizations can substantially reduce their risk exposure while maintaining Redis's performance advantages. Our research contributes to the theoretical understanding of NoSQL security architectures and practical knowledge for securing Redis deployments. Hence, we advance the state of database security and protect the critical data infrastructure that powers modern digital services.

Author Contributions: Conceptualization: M.N.M. and M.E.I.; Methodology: M.N.M.; Software: M.N.M.; Validation: M.N.M. and M.E.I.; Analysis: M.N.M.; Investigation: M.N.M.; Resources: M.N.M.; Writing—original draft preparation: M.N.M.; Writing—review and editing:

M.N.M. and M.E.I.; Supervision: M.E.I.; Project administration: M.N.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: We would like to express gratitude to the Department of Computer Science, Nigerian Defence Academy, Kaduna, for providing us with the necessary support to conduct the research.

Conflicts of Interest: The authors declare that they have no financial or non-financial interests that could be perceived as influencing the work described in this manuscript

References

[1] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer (Long. Beach. Calif).*, vol. 45, no. 2, pp. 30–36, Feb. 2012, doi: 10.1109/MC.2011.389.

[2] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *J. Cloud Comput. Adv. Syst. Appl.*, vol. 2, no. 1, p. 22, Dec. 2013, doi: 10.1186/2192-113X-2-22.

[3] H. B. S. Reddy, R. R. S. Reddy, R. Jonnalagadda, P. Singh, and A. Gogineni, "Analysis of the Unexplored Security Issues Common to All Types of NoSQL Databases," *Asian J. Res. Comput. Sci.*, pp. 1–12, May 2022, doi: 10.9734/ajrcos/2022/v14i130323.

[4] S. Sicari, A. Rizzardi, and A. Coen-Porisini, "Security&privacy issues and challenges in NoSQL databases," *Comput. Networks*, vol. 206, p. 108828, Apr. 2022, doi: 10.1016/j.comnet.2022.108828.

[5] D. Van Landuyt, V. Wijshoff, and W. Joosen, "A study of NoSQL query injection in Neo4j," *Comput. Secur.*, vol. 137, p. 103590, Feb. 2024, doi: 10.1016/j.cose.2023.103590.

[6] OWASP Foundation, "OWASP Top Ten," *owaps.org*, 2022. https://owasp.org/www-project-top-ten/

[7] S. Patil, M. Rao, L. Misal, D. Phaldesai, and K. Shivsharan, "A Review of the OW ASP Top 10 Web Application Security Risks and Best Practices for Mitigating These Risks," in *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, Aug. 2023, pp. 1–8. doi: 10.1109/ICCUBEA58933.2023.10392030.

[8] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[9] X. Qi, H. Hu, X. Wei, C. Huang, X. Zhou, and A. Zhou, "High Performance Design for Redis with Fast Event-Driven RDMA RPCs," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12112 LNCS, 2020, pp. 195–210. doi: 10.1007/978-3-030-59410-7_12.

[10] J. Zablocki, M. Daniel, C. Shantanu, S. Pramod, and L. Jianbo, "Real-time detection and clustering of emerging fraud patterns," 2021 [Online]. Available: https://patents.justia.com/patent/10938853

[11] DB-Engines, "DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems," *DB-Engines*. 2024. [Online]. Available: https://db-engines.com/en/

[12] G. Kaur and J. Kaur, "In-Memory Data processing using Redis Database," *Int. J. Comput. Appl.*, vol. 180, no. 25, pp. 26–31, Mar. 2018, doi: 10.5120/ijca2018916589.

[13] V. C. Hu, "Access control on NoSQL databases," May 2024. doi: 10.6028/NIST.IR.8504.

[14] X. Chen, J. Jiang, W. Zhang, and X. Xia, "Fault Diagnosis for Open Source Software Based on Dynamic Tracking," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, Nov. 2020, pp. 263–268. doi: 10.1109/DSA51864.2020.00047.

[15] V. Das, *Learning Redis*. Packt Publishing Ltd, 2015.

[16] Q. Castro, "Security Advisory: CVE-2024-31449, CVE-2024-31227, CVE-2024-31228," *Redis.io*, 2024. https://redis.io/blog/security-advisory-cve-2024-31449-cve-2024-31227-cve-2024-31228/

[17] A. Johns, "Redis Injection Vulnerabilities in LLM-Powered RAG Systems," *Secure Cortex Blog*, 2024. https://blog.securecortex.com/2024/10/large-language-models-injections-in-rag.html

[18] E. Ankomah *et al.*, "A Comparative Analysis of Security Features and Concerns in NoSQL Databases," in *Communications in Computer and Information Science*, vol. 1726 CCIS, 2022, pp. 349–364. doi: 10.1007/978-981-19-8445-7_22.

[19] B. Hou, K. Qian, L. Li, Y. Shi, L. Tao, and J. Liu, "MongoDB NoSQL Injection Analysis and Detection," in *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, Jun. 2016, pp. 75–78. doi: 10.1109/CSCloud.2016.57.

[20] N. Gupta and R. Agrawal, "NoSQL Security," in *Advances in Computers*, vol. 109, Academic Press Inc., 2018, pp. 101–132. doi: 10.1016/bs.adcom.2018.01.003.

[21] V. Sachdeva, "Vulnerability Assesment For Advanced Injection Attacks Against Mongodb," *J. Mech. Contin. Math. Sci.*, vol. 14, no. 1, pp. 402–413, Feb. 2019, doi: 10.26782/jmcms.2019.02.00028.

[22] S. Dwivedi, R. Balaji, P. Ampatt, and S. D. Sudarsan, "A Survey on Security Threats and Mitigation Strategies for NoSQL Databases," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 14424 LNCS, 2023, pp. 57–76. doi: 10.1007/978-3-031-49099-6_4.

[23] D. Fiser, "More Than 8,000 Unsecured Redis Instances Found in the Cloud," *Trend Micro*, 2020. https://www.trendmicro.com/en_us/research/20/d/more-than-8-000-unsecured-redis-instances-found-in-the-cloud.html?_ga=2.23954494.54084514.1736434217-1683098876.1736259621

[24] C. Carlos, M. Steven, and R. Peter, *Database System: Design, Implementation, and Management*. 2018.

[25] T. Macedo and F. Oliviera, *Redis Cookbook*, First. O'Reilly Media Inc,., 2011.

[26] R. Rao, "What Databaseless (DBLess) Architecture Is—and Why It's the Future," *Redis.io*. 2021. [Online]. Available: https://redis.io/blog/dbless-architecture-and-why-its-the-future/

[27] D. Eddelbuettel, "Redis for Market Monitoring," *arXiv*. Mar. 15, 2022. [Online]. Available: http://arxiv.org/abs/2203.08323

[28] G. Muradova, M. Hematyar, and J. Jamalova, "Advantages of Redis in-memory database to efficiently search for healthcare medical supplies using geospatial data," in *2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT)*, Oct. 2022, pp. 1–5. doi: 10.1109/AICT55583.2022.10013544.

[29] R. Ajeet, "Top 5 Reasons Why DevOps Teams Love Redis Enterprise," *Redis.io*, 2020. https://redis.io/blog/why-devops-teams-love-redis-enterprise/

[30] T. Fiebig, A. Feldmann, and M. Petschick, "A One-Year Perspective on Exposed In-memory Key-Value Stores," in *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, Oct. 2016, pp. 17–22. doi: 10.1145/2994475.2994480.

[31] R. A. G. Sanchez, D. J. M. Bernal, and H. D. J. Parada, "Security assessment of Nosql Mongodb, Redis and Cassandra database managers," in *2021 Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI)*, Sep. 2021, pp. 1–7. doi: 10.1109/CONIITI53815.2021.9619597.

[32] Asadulla Khan Zaki and Indiramma M., "A novel redis security extension for NoSQL database using authentication and encryption," in *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, Mar. 2015, pp. 1–6. doi: 10.1109/ICECCT.2015.7226101.

[33] A. Costin, "Lua Code: Security Overview and Practical Approaches to Static Analysis," in *2017 IEEE Security and Privacy Workshops (SPW)*, May 2017, vol. 2017-Decem, pp. 132–142. doi: 10.1109/SPW.2017.38.

[34] A. Stasinopoulos, C. Ntantogian, and C. Xenakis, "Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications," *Int. J. Inf. Secur.*, vol. 18, no. 1, pp. 49–72, Feb. 2019, doi: 10.1007/s10207-018-0399-z.

[35] S. Kairoju, R. Sultana, and P. Danidharia, "Security Audit of NoSQL DBMS," *ERA: Education and Research Archive*. 2021. [Online]. Available: https://era.library.ualberta.ca/items/6b114eb6-3c87-4db5-8571-1cb3a5fb6cb1/download/13d0ae5c-57b6-4cc4-9f12-5e66a706b579

[36] R. M. A. and N. H. Ashwaq A. Alotaibi, Reem M. Alotaibi and Nermin Hamza, Ashwaq A. Alotaibi, "Access Control Models in NoSQL Databases: An Overview," *J. King Abdulaziz Univ. Comput. Inf. Technol. Sci.*, vol. 8, no. 1, pp. 1–9, Mar. 2019, doi: 10.4197/Comp.8-1.1.

[37] U. Saxena and S. Sachdeva, "An Insightful View on Security and Performance of NoSQL Databases," in *Communications in Computer and Information Science*, vol. 799, 2018, pp. 643–653. doi: 10.1007/978-981-10-8527-7_54.

[38] K. Fahd, S. Venkatraman, and F. Khan Hammeed, "A Comparative Study of NOSQL System Vulnerabilities with Big Data," *Int. J. Manag. Inf. Technol.*, vol. 11, no. 4, pp. 1–19, Nov. 2019, doi: 10.5121/ijmit.2019.11401.

[39] A. Nikiforova, A. Daskevics, and O. Azeroual, "NoSQL Security: Can My Data-driven Decision-making Be Influenced from Outside?," in *Big Data and Decision-Making: Applications and Uses in the Public and Private Sector*, Emerald Publishing Limited, 2023, pp. 59–73. doi: 10.1108/978-1-80382-551-920231005.

[40] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *College of Computing Georgia Institute of Technology*, 2008. https://faculty.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf

[41] N. Yaakov and O. Itach, "New Redis Backdoor Malware," *Aqua Nautilus Discovers Redigo*, 2022. https://www.aquasec.com/blog/redigo-redis-backdoor-malware/

[42] National Institute of Standards Technology (NIST), "NVD - CVE-2022-0543," *National Vulnerability Database (NVD)*, 2022. https://nvd.nist.gov/vuln/detail/cve-2022-0543

[43] National Institute of Standards Technology (NIST), "NVD - CVE-2024-46981," *National Vulnerability Database*, 2024.

[44] National Institute of Standards Technology (NIST), "NVD - CVE-2021-29477," *National Vulnerability Database*, 2021. https://nvd.nist.gov/vuln/detail/cve-2021-29477

[45] National Institute of Standards Technology (NIST), "NVD - CVE-2021-32762," *National Vulnerability Database*, 2021. https://nvd.nist.gov/vuln/detail/cve-2021-32762

[46] D. Fiser and J. Horejsi, "Exposed Redis Instances Abused for Remote Code Execution, Cryptocurrency Mining," *Trend Micro*, 2020. https://www.trendmicro.com/en_us/research/20/d/exposed-redis-instances-abused-for-remote-code-execution-cryptocurrency-mining.html

[47] National Institute of Standards Technology (NIST), "NVD - CVE-2021-32627," *National Vulnerability Database*, 2021. https://nvd.nist.gov/vuln/detail/cve-2021-32627

[48] National Institute of Standards Technology (NIST), "NVD - CVE-2020-4670," *National Vulnerability Database*, 2020. https://nvd.nist.gov/vuln/detail/cve-2020-4670

[49] Forum of Incident Response and Security Teams (FIRST), "Common Vulnerability Scoring System version 3.1: Specification Document," *first.org*, 2019. https://www.first.org/cvss/v3-1/specification-document

[50] W. H. Douglas and R. Seiersen, *How to Measure Anything in Cybersecurity Risk*. John Wiley & Sons, Inc, 2023. [Online]. Available: https://www.wiley.com/en-us/How+to+Measure+Anything+in+Cybersecurity+Risk%2C+2nd+Edition-p-9781119892304