Rarimo – Backend WebApp Pentest

Rarimo

/ DRAFT /

HALBORN

Rarimo – Backend WebApp Pentest - Rarimo

Prepared by: HALBORN

Last Updated 03/16/2024

Date of Engagement by: February 26th, 2024 - March 8th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	0	0	2	0

TABLE OF CONTENTS

1. Introduction

Rarimo engaged Halborn to conduct a security assessment on their web application backend. The security assessment was scoped to the codebase provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to verify the security of the backend. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that backend functions operate as intended

- Identify potential security issues within the backend

The code provided to Halborn did not present any vulnerability directly exploitable by an attacker. However, it was identified that no intense data validation was performed on the backend. The data retrieved from the users, it was converted to different data types, triggering errors in case the data was incorrect, which acted as an indirect data validation.

Furthermore, an incorrect declaration of a switch statement was found in the code. The statement did not contain a default clause that would catch the program flow in case the available options were not satisfied by the variable.

Lastly, multiple vulnerable dependencies were found present on the code, it is recommended to update them to the last version available or reduce the attack surface as much as possible.

In summary, Halborn identified some security recommendations that were successfully applied by the Rarimo team.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Mapping Application Content and Functionality.

- Technology stack-specific vulnerabilities and Code Assessment.

- Known vulnerabilities in 3rd party / OSS dependencies.

- Application Logic Flaws.

- Authentication / Authorization flaws.

- Input Handling.

- Fuzzing of all input parameters.

- Testing for different types of sensitive information leakages: memory, clipboard, etc.

- Test for Injection (SQL/JSON/HTML/JS/Command/Directories...).

- Brute Force Attempts.

- Perform static analysis on code.

- Ensure that coding best practices are being followed by the Rarimo team.

- Technology stack-specific vulnerabilities and Code Assessment.

- Known vulnerabilities in 3rd party / OSS dependencies.

- Identify potential vulnerabilities that may pose a risk.

4. Manual Testing

- **Request Parameters Verification**Input validation is a critical security control that involves scrutinizing data received from users or external systems to confirm that it adheres to the anticipated format, type, and value constraints prior to any processing. This precautionary measure is crucial for safeguarding against prevalent security threats, including SQL injection, cross-site scripting (XSS), and buffer overflow attacks. By ensuring that only properly formatted and expected input is processed, systems can significantly reduce the risk of exploitable vulnerabilities that could lead to unauthorized access, data leaks, or service disruptions.
- **Hardcoded Credentials**Embedding sensitive information, like usernames, passwords, or API keys, directly in source code is a risky practice. It exposes critical data to potential compromise if the code becomes accessible to unauthorized parties. This method of handling secrets threatens the security of the system and undermines data privacy by placing sensitive credentials at risk of exposure. Such practices should be avoided in favor of more secure methods of managing and accessing sensitive information, such as environment variables or secure vaults, to enhance overall application security and data protection.
- **Sensitive Error Logging**Accidentally recording sensitive information—ranging from personal details and credentials to critical system configurations—in application logs poses a substantial security risk. This oversight can lead to unintended information disclosure, offering malicious actors valuable insights that could be exploited to compromise the system or access protected data. Such leaks breach privacy norms and elevate the risk of targeted attacks by providing adversaries with the knowledge needed to navigate the system’s defenses or execute sophisticated exploits. It's imperative to implement stringent logging policies and review mechanisms to prevent sensitive data from being logged, thereby safeguarding the integrity and confidentiality of the system's information.
- **Incorrect Error Handling**Inadequate management of error conditions in an application can precipitate crashes, unpredictable behavior, or security vulnerabilities. Effective error handling is crucial for intercepting and addressing errors in a manner that prevents the disclosure of sensitive information and maintains the application's stability. By implementing comprehensive error handling mechanisms, developers can ensure that errors are not only caught but are also handled gracefully, thereby averting potential system disruptions or exploitations. This practice is essential for preserving the integrity and reliability of the application, enhancing user experience, and fortifying the application against attacks that exploit unhandled errors.
- **Uncontrolled Panics**Within programming environments such as Go, an uncontrolled panic refers to a situation where an application experiences a runtime error that is not appropriately intercepted or managed, leading to an abrupt termination of the application. Such panics are critical events that disrupt the normal flow of execution, often resulting from unforeseen errors like accessing a nil pointer or attempting an out-of-bounds array access. Implementing structured error handling and panic recovery mechanisms is essential to safeguard against these uncontrolled terminations. By employing techniques such as deferred functions with recovery logic, developers can capture and handle panics, allowing the application to continue running or to terminate gracefully. This approach enhances application robustness and reliability and ensures a better user experience by avoiding sudden crashes.
- **Outdated Dependencies Review**The practice of periodically reviewing and updating the external libraries or dependencies an application relies on is crucial for maintaining its security and functionality. Outdated dependencies are a significant risk factor, as they may harbor known

vulnerabilities or bugs that attackers could leverage to compromise the application. This process, often part of a broader application security strategy, helps ensure that the software stays protected against known attack vectors by incorporating the latest security patches and improvements provided by dependency maintainers. Regularly updating dependencies minimizes the attack surface and ensures that the application benefits from performance enhancements and new features, thereby maintaining its integrity and resilience against potential security threats.

- **API Testing**API testing involves directly interacting with application programming interfaces (APIs) to validate their correctness, performance under varied scenarios, and graceful error handling capabilities. This form of testing is critical for ensuring that APIs meet their design specifications, providing reliable and secure communication between different software components. It encompasses a comprehensive assessment of both security aspects, such as authentication, authorization, and data encryption, and functional elements, including response times, data accuracy, and handling of valid or invalid input. By thoroughly testing APIs, developers can identify and address potential vulnerabilities or functional inconsistencies early in the development cycle, enhancing the overall robustness and security of the application.
- **Use of Insecure Functions**The use of functions or methods known to be vulnerable or insecure poses significant risks to application security. Such vulnerabilities may stem from inherent flaws in the design of these functions or from their failure to implement necessary security checks. Common examples include reliance on deprecated cryptographic algorithms, which are no longer considered secure against modern attack techniques, or functions that fail to sanitize user input adequately, leaving the application susceptible to injection attacks. Employing these insecure functions can inadvertently introduce exploitable weaknesses into the system, making it easier for attackers to compromise data integrity, confidentiality, or availability. Developers are advised to stay abreast of current security best practices and recommendations, replacing or updating any insecure functions with secure alternatives that offer robust protection against potential threats.
- **Incorrect Route Handling**Improper management of web application routes or endpoints can create significant security vulnerabilities, potentially resulting in unauthorized access or disclosure of sensitive information. Such vulnerabilities can arise from various issues, including open redirects, which can trick users into visiting malicious sites; inadequate access controls, allowing unauthorized users to access restricted areas; and incorrect handling of HTTP methods, leading to unintended application behaviors. Effective route handling is crucial for ensuring that each endpoint operates as intended, securely processing requests and responding with appropriate information. By rigorously defining and enforcing access controls, validating and sanitizing inputs, and correctly configuring HTTP method responses, developers can mitigate risks and safeguard the application against common attack vectors, thus protecting both the application's integrity and its users' data.
- **Incorrect parameterized request**Properly parameterized queries are a fundamental security practice in database management, wherein placeholders are used for user inputs, ensuring that these inputs are processed solely as data and not interpreted as SQL commands. This method serves as a critical defense mechanism against SQL injection attacks, a prevalent threat where attackers exploit vulnerabilities to execute malicious SQL commands. Incorrect management of query parameters can undermine this defense, exposing the system to risks where attackers could alter or retrieve data without authorization. By adhering to best practices for parameterized queries, developers can significantly enhance the security of database operations, preventing unauthorized access and manipulation of sensitive information.

5. RISK METHODOLOGY

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

- **10** - CRITICAL
- **9 - 8** - HIGH
- **7 - 6** - MEDIUM
- **5 - 4** - LOW
- **3 - 1** - VERY LOW AND INFORMATIONAL

6. SCOPE

FILES AND REPOSITORY

(a) Repository: passport-identity-provider

(b) Assessed Commit ID: c13e386

(c) Contracts in scope:

Out-of-Scope:

REMEDIATION COMMIT ID:

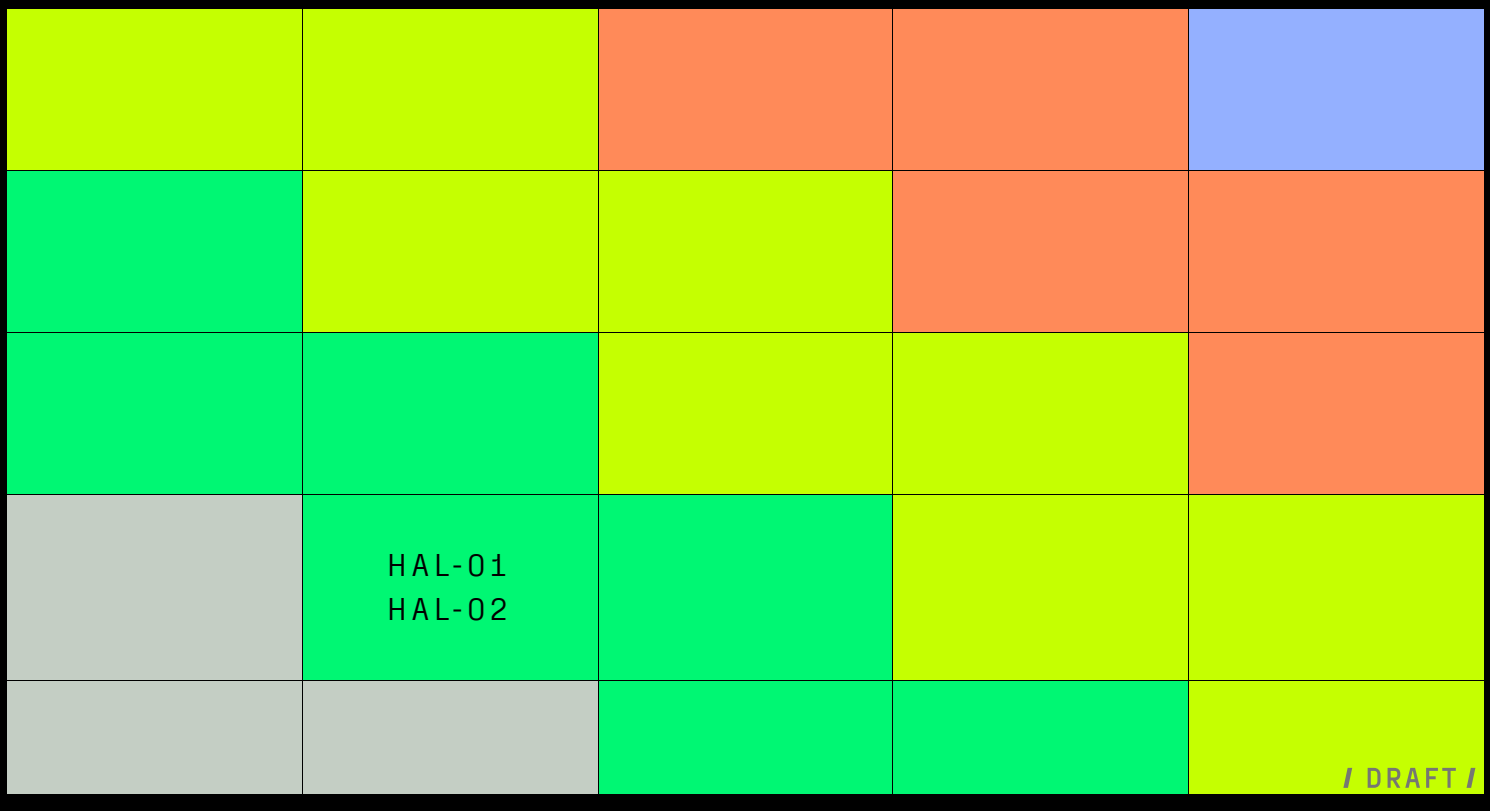
- 8f2ab7b

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	0

IMPACT X LIKELIHOOD



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - SWITCH CLAUSE WITH NO DEFAULT OPTION	Low	SOLVED - 03/12/2024
HAL-02 - MULTIPLE OUTDATED DEPENDENCIES	Low	SOLVED - 03/12/2024

8. FINDINGS & TECH DETAILS

8.1 (HAL-01) SWITCH CLAUSE WITH NO DEFAULT OPTION

// LOW

Description

The switch statement allows for a series of case checks against a single expression's value. Typically, this structure also allows for a default case, which serves as a fallback when none of the explicitly listed cases match the value of the expression.
The absence of a default case means that the switch structure does not provide a specific course of action when none of the cases match the given value.

Proof of Concept

```
switch algorithms[req.Data.DocumentSOD.Algorithm] {
    case SHA1withECDSA:
        if err := verifier.VerifyGroth16(req.Data.ZKProof,
cfg.VerificationKeys[SHA1]); err != nil {
            Log(r).WithError(err).Error("failed to verify Groth16")
            ape.RenderErr(w, problems.BadRequest(err)...)
            return
        }
    case SHA256withRSA, SHA256withECDSA:
        if err := verifier.VerifyGroth16(req.Data.ZKProof,
cfg.VerificationKeys[SHA256]); err != nil {
            Log(r).WithError(err).Error("failed to verify Groth16")
            ape.RenderErr(w, problems.BadRequest(err)...)
            return
        }
    }
```

Score

Impact: 2

Likelihood: 2

Recommendation

Implement a `default` case as a fallback when none of the explicitly listed cases match the value of the expression.

Remediation Plan

SOLVED: The **Rarimo team** added a default clause into the switch statement as part of their coding approach. This inclusion ensures that the switch statement has a fallback option to execute in cases where none of the specified cases match the input.t.

Remediation Hash

8f2ab7b40863bf1c2b2ca0bbe74ee130064e48c7

8.2 (HAL-02) MULTIPLE OUTDATED DEPENDENCIES

// LOW

Description

During the security assessment, an automated check was performed against the project dependencies. The command `go list -u -m -json all | go-mod-outdated -update -direct` revealed the use of multiple vulnerable or outdated dependencies.
Disclaimer: During the assessment, not direct impact was found related to the outdated packages.

Proof of Concept

```
+-----------------------------------------------+-----------------------------------
+-----------------------------------+
|                    MODULE                     |              VERSION
|            NEW VERSION             |
+-----------------------------------------------+-----------------------------------
+-----------------------------------+

github.com/Masterminds/squirrel               | v1.4.0                            |
v1.5.4
github.com/ethereum/go-ethereum               | v1.11.5                           |
v1.13.14
github.com/go-ozzo/ozzo-validation/v4         | v4.2.1                            |
v4.3.0
github.com/google/uuid                        | v1.3.0                            |
v1.6.0
github.com/iden3/go-iden3-crypto              | v0.0.15                           |
v0.0.16
github.com/imroc/req/v3                       | v3.42.3                           |
v3.43.0
github.com/rarimo/certificate-transparency-go | v0.0.0-20240216144634-4291bc43f73b |
v0.0.0-20240305114501-050b1f19639a
gitlab.com/distributed_lab/figure             | v2.1.0+incompatible               |
v2.1.2+incompatible
gitlab.com/distributed_lab/figure/v3          | v3.1.3                            |
v3.1.4
gitlab.com/distributed_lab/kit                | v1.11.2                           |
v1.11.3
```

Score

Impact: 2

Likelihood: 2

Recommendation

Update to the latest version available.

Remediation Plan

SOLVED: The **Rarimo team** updated all the outdated dependencies to the latest version available.

Remediation Hash

8f2ab7b40863bf1c2b2ca0bbe74ee130064e48c7

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.