

Navi Token

Audit

Presented by:



OtterSec contact@osec.io

Michał Bochnak embe221ed@osec.io



Contents

01 Executive Summary **2**
Overview 2
Key Findings 2
02 Scope **3**
03 Findings **4**
04 General Findings **5**
OS-NAV-SUG-00 | Hardcoded Recipient In Init 6

Appendices

A Vulnerability Rating Scale **7**
B Procedure **8**

© 2023 Otter Audits LLC. All Rights Reserved.

01 | Executive Summary

Overview

Navi engaged OtterSec to assess the `token` program. This assessment was conducted between January 19th and January 19th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 1 suggestion throughout this audit engagement.

© 2023 Otter Audits LLC. All Rights Reserved.

02 | Scope

The source code was delivered to us in a git repository at github.com/naviprotocol/protocol-core. This audit was performed against commit 0436bcc.

A brief description of the programs is as follows:

Name	Description
token	Module responsible for creating Navi token currency and transferring the `TreasuryCap`.

© 2023 Otter Audits LLC. All Rights Reserved.

03 | Findings

Overall, we reported 1 finding.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.

© 2023 Otter Audits LLC. All Rights Reserved.

04 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-NAV-SUG-00	The `init` function sends the created `TreasuryCap` to the hardcoded address

© 2023 Otter Audits LLC. All Rights Reserved.

OS-NAV-SUG-00 | Hardcoded Recipient In Init

Description

In `Recipient: address` constant represents a hardcoded address that will receive the `treasury_cap` for a token created during module initialization. This approach seems unusual, and we believe it may be safer to send the `treasury_cap` to the sender, who can later send it to the desired recipient.

navi.move RUST

```
const Recipient: address = @0x01; // **You need to specify an address to publish

// ... consts

fun init(witness: NAVI, ctx: &mut TxContext) {
    // ... code

    let (treasury_cap, metadata) = coin::create_currency<NAVI>(
        // ... args
    );
    // ... code

    transfer::public_transfer(treasury_cap, Recipient);
}
```

Remediation

Transfer the newly created `treasury_cap` to the sender in the `init` function. The `treasury_cap` can later be sent to the desired recipient using the `public_transfer` function.

Patch

Resolved in 990e9c4.

© 2023 Otter Audits LLC. All Rights Reserved.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

© 2023 Otter Audits LLC. All Rights Reserved.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2023 Otter Audits LLC. All Rights Reserved.