Least Authority
PRIVACY MATTERS

ERTP + Zoe

# Security Audit Report: Vulnerability Assessment

# Agoric Systems

Final Audit Report: 22 December 2021

# Table of Contents

# Overview

## Background

Agoric Systems requested that Least Authority perform a vulnerability assessment of the following components:

- **Electronic Rights Transfer Protocol (ERTP)**: Agoric's digital asset standard and a uniform way of transferring tokens and other digital assets in JavaScript. All kinds of digital assets can be easily created and can be transferred in exactly the same ways, with exactly the same security properties.
- **Zoe**: a framework for building smart contracts including, but not limited to, auctions, swaps, and decentralized exchanges. Zoe itself is a smart contract written in JavaScript and running on the Agoric platform.

Agoric is an open-source development company launching an interoperable Proof-of-Stake chain and economy. Agoric's JavaScript-native smart contract platform aims to offer developers a safe, reusable library of Decentralized Finance (DeFi) components to rapidly build and deploy on-chain.

## Project Dates

- **August 9 - September 3**: Code review *(Completed)*
- **September 8**: Delivery of Initial Audit Report *(Completed)*
- **September 27 - October 22**: Additional Code review *(Completed)*
- **October 27**: Delivery of Updated Initial Audit Report *(Completed)*
- **November 13 - December 21**: Verification Review *(Completed)*
- **December 22**: Final Audit Report delivered *(Completed)*

## Review Team

- David Braun, Security Researcher and Engineer
- Dominc Tarr, Security Researcher and Engineer
- Florian Detig, Security Researcher and Engineer
- Jehad Baeth, Security Researcher and Engineer
- Wanas ElHassan, Security Researcher and Engineer

# Coverage

## Target Code and Revision

For this audit, we performed research, investigation, and review of ERTP + Zoe followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:
- ERTP: https://github.com/Agoric/agoric-sdk/tree/master/packages/ERTP
- Zoe: https://github.com/Agoric/agoric-sdk/tree/master/packages/zoe

Specifically, we examined the following Git revision for our initial review:

```
198e5c37b4ce3738ed5776c36c949847a226265c
```

For the verification, we examined the Git revision:

`fe9a9ff3de86608a0b1f8f9547059f89d45b948d`

In addition, an issue was identified by the Agoric team in a later commit of the repository. Our team reviewed the issue and the implemented mitigation:

- https://github.com/Agoric/agoric-sdk/commit/b67bfcb9051cdcf780aff1a10653635448b21eae#diff-fe679b1c51464cc0f59ad0c778d5c879cf52dfcb78635b8a2f263f5ea57ca3dc

For the review, the repository was cloned for use during the audit and for reference in this report:

- https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP

All file references in this document use Unix-style paths relative to the project's root directory.

Finally, the Zoe smart contracts and the Fees and Metering feature, along with any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

## Supporting Documentation

The following documentation was available to the review team:

- ERTP:
    - ERTP `README.md`: https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP/blob/master/packages/ERTP/README.md
    - ERTP Introduction: https://agoric.com/documentation/getting-started/ertp-introduction.html
    - ERTP Guide: https://agoric.com/documentation/ertp/guide/
- ERTP Dependencies:
    - Assert: https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP/blob/master/packages/assert
    - Marshal: https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP/blob/master/packages/marshal
    - Notifier: https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP/blob/master/packages/notifier
- Zoe:
    - Zoe `README.md`: https://github.com/LeastAuthority/Agoric-Systems-Zoe-ERTP/blob/master/packages/zoe/README.md
    - Zoe Introduction: https://agoric.com/documentation/getting-started/intro-zoe.html
    - Zoe Guide: https://agoric.com/documentation/zoe/guide/
- Agoric Documentation Guide: https://agoric.com/documentation/getting-started/
- Algebraic Properties of Amounts: https://github.com/Agoric/agoric-sdk/issues/557
- Agoric JavaScript Programming: https://agoric.com/documentation/guides/js-programming/
- User Documentation for Vats: https://agoric.com/documentation/guides/js-programming/vats.html
- User Documentation for Far() and Remotable Objects: https://agoric.com/documentation/guides/js-programming/far.html
- User Documentation for Eventual Send: https://agoric.com/documentation/guides/js-programming/eventual-send.html
- User Documentation for Notifiers and Subscriptions: https://agoric.com/documentation/guides/js-programming/notifiers.html

- User Documentation for SES: https://agoric.com/documentation/guides/js-programming/ses/ses-guide.html
- Zoe/ERTP Audit Overview.pdf (*shared with Least Authority via Discord 9 August 2021*)
- [Draft] -- Purple Team Report, Hardened JavaScript.pdf (*shared with Least Authority via email 27 September 2021*)
- ERTP_Zoe-- Scope + Documentation 2.pdf (*shared with Least Authority via email 27 September 2021*)
- Handling of Authority in ERTP and Zoe Google Document (*shared with Least Authority via email 1 October 2021*)
- Moar Agoric Audit Info Google Slide Presentation (*shared with Least Authority via email 1 October 2021*)
- Article "*Preventing Reentrancy Attacks in Smart Contracts*": https://medium.com/agoric/preventing-reentrancy-attacks-in-smart-contracts-3899bf837f23
- Agoric + Protocol Labs // Higher Order Smart Contracts across Chains by Mark S. Miller YouTube Video: https://www.youtube.com/watch?v=4gNOOC6vucY&list=PLKr-mvz8uvUgybLg53lgXSeLOp4BiwvB2&t=1860s
- Hack the Orb - Day 5 Events (Nov 11) YouTube Video: https://youtu.be/jHAi7FsbB9Y?t=3606
- M. Miller, T. Van Cutsem, B. Tulloh, 2013, "Distributed Electronic Rights in JavaScript." *ESOP'13 22nd European Symposium on Programming, Springer*
- M. Miller, E.D. Tribble, J. Shapiro, 2005, "Concurrency Among Strangers Programming in E as Plan Coordination." *TGC 2005, LNCS 3705, pp. 195–229*.
- S. Drossopoulou, J. Noble, T. Murray, M. Miller, 2015, "Reasoning about Risk and Trust in an Open World." Victoria University of Wellington.

In addition, this audit report references the following document:

- M. Samuel Miller, 2006, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control." [M06]

## Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution of the code;
- Vulnerabilities in the code;
- Protection against malicious attacks and other ways to exploit the code;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

# Findings

## General Comments

The ERTP and Zoe implementations use an object capability security model (OCAP), which is a novel approach in the field of blockchains and distributed systems. Agoric utilizes Hardened JavaScript (previously known as SES, or Secure ECMAScript) as the smart contract implementation language. Hardened JavaScript is a runtime library that allows running untrusted code securely. In a previous security audit (not published at the time of delivering this Final Audit Report), our team performed review and analysis of Agoric's Hardened JavaScript and did not identify any attacks that violated the immutability of hardened JavaScript objects.

In addition to Hardened JavaScript, Agoric's ERTP and Zoe rely on multiple layers of functionality that are required for the system to function as intended and are assumed to behave correctly. The Agoric system utilizes the Tendermint consensus engine that provides the consensus layer. SwingSet is the virtual machine that provides a JavaScript runtime environment and divides the execution environment up into isolated vats. Vats are a level of abstraction that aim to allow secure interaction across the various layers of the system. Within a particular vat, objects and functions can communicate with each other synchronously. In addition to creating vats, SwingSet handles communications between individual vats and between vats and external units, including the Agoric consensus layer. SwingSet utilizes a Cosmos SDK module to interact with the consensus layer.

## System Design

Our team performed a manual code review of the Zoe and ERTP implementations, in addition to a close study of their respective system designs. We found that ERTP is built upon a relatively small number of concepts and is sufficiently easy to understand. The project documentation provided by the Agoric team highlights Zoe's composability of secure programs, which means that each individual part is simple and easy to understand, and that more sophisticated programs are built through the combination of the simple programs.

Following comprehensive review and analysis, and once we became familiarized with the code and conventions, we gained a clear understanding of the security benefits of this modular design because of the carefully delineated security capabilities. The capabilities are independently defined and tested, then aggregated in higher order functions, before being exported in bundles from the various modules. This approach simplifies the process of verifying that each module has an appropriate level of authority for its purpose and how much authority it is exposing in its public interface.

Additionally, key parts of the system (e.g. Zoe and user-defined smart contracts) are isolated from each other in vats to partition risk using Hardened JavaScript. While Hardened JavaScript has been worked on for years by qualified and experienced security engineers, in addition to undergoing some battle testing, it is still new and largely unproven.

Due to the immutable nature of hardened objects and the lack of shared state, the code is referentially transparent.This allowed our team to properly isolate concerns when reviewing a certain piece of code. Most of the code is an API that gives access to hardened values. Knowing that values are immutable allows reasoning about security beyond the concern of a tampered state as part of smart contract execution. The execution model is capability-based, as the only abilities a smart contract can have are those which the Zoe seat affords them, and that, itself, is passed to the smart contract call. Analogous to other blockchain systems, this gives the smart contract view of its state and what a smart contract is able to do.

## Investigation of Security Properties

During this audit, we focused closely on investigating the below security properties, which the Agoric team has stated to be guaranteed by the system.

### Zoe Contract Facet

The Zoe Contract Facet (ZCF) is an API for the Zoe Service that is provided to each deployed smart contract. We investigated the security property that smart contract code should not be able to adversely affect the functioning of the ZCF, despite being within the same vat. Specifically, we attempted to violate rights conservation, which is the property that ensures that no funds are lost or gained upon seat reallocations. We also attempted to violate offer safety, which is the property that a user can always get either a full refund or the item that an offer is made on. Both properties are enforced in ZCF, and we did not identify instances where a smart contract is able to violate either.

In addition, we examined all of the ZCF endpoints for potential methods of abuse. We looked for ways that a smart contract could maliciously gain access to the internals of ZCF, such as `activeZCFSeats`, and cause harm. We were unable to identify an attack that would disrupt the expected behavior of ZCF using Zoe smart contract code. Furthermore, our team examined the code in `rightsConservation.js`, `startInstance.js`, `zcfSeat.js`, `offerSafety.js`, and `zcfZygote.js`. We attempted to violate rights conservation and offer safety through functions, such as `reallocateForZCFMint` and others. We were unable to violate these properties.

#### Zoe Service

Our team investigated and attempted to violate the security property that smart contract code must not be able to adversely affect the functionality of the Zoe Service. We were unable to violate this security property. We investigated the invitation handler supplied by a smart contract, a potential source of external behavior and did not identify misbehaviour of this function by which Zoe's vat can be impacted.

#### Zoe Escrow

The Zoe Service functions as an asset custodian and a proxy for all asset transfers. We investigated the security property that Zoe smart contracts do not have access to user assets held by the Zoe Service in escrow. We examined the code in `escrowStorage.js`, `zoeStorageManager.js`, `rightsConservation.js`, `zoe.js` and others. We were unable to identify a way for smart contracts to directly access user assets held in escrow by the Zoe Service.

#### Potential Race Conditions

We checked `paymentLedger.js` and found that the only interaction with the underlying `scalarWeakMap` is happening within a synchronous call, which means only one of the operations would be able to interact with the storage. No race conditions or interleaving scenarios were identified in `paymentLedger` or issuer-related functionality. However, race conditions within ZCF calls (e.g. `mintGains` or `burnLosses`) could create inconsistent states between a smart contract and Zoe. We recommend chaining promises which update state with `.then()` to remove nondeterminism from ordering (Issue C). Asynchronous calls to different remotables provide no guarantee on the execution order, which can lead to Zoe being in an unintended inconsistent state with the smart contract.

#### Make Invitation

We investigated the security property that a user can only receive an invitation to a smart contract instance if the Zoe smart contract code calls the function `zcf.makeInvitation`. The trusted issuer for invitations is created early on in the bootstrapping phase of the ZCF vat. That issuer is accessible only to the functions within the invitation kit, which uses the invitation issuer's mint function. Due to the private visibility of this issuer, our team could not identify an instance where an invitation can be created in a way other than calling `zcf.makeInvitation`.

#### Misbehaving Issuers

In ERTP, issuers are authorities that keep track of who owns digital assets. For example, there may be an issuer for Bitcoin that keeps track of everyone's BTC balances on the platform. We investigated ways in which a misbehaving issuer could adversely affect the Zoe Service.

##### Make Empty Purse

We examined the impact on Zoe if a misbehaving issuer were to cause the creation of a *purse* (ERTP's version of a wallet) to hang or fail. We found a negative impact on the `startInstance` and `saveIssuer` functions (Suggestion 2).

**Getting Brand & Display Information**
We examined if a malicious issuer could prevent access to brand information and `displayInfo` causing a hang or failure. We reviewed the code in `startInstance.js`, `zoeStorageManager.js`, and `issuerStorage.js`. We traced possible failures in getting the brand and `displayInfo` in the `storeIssuer` function up through invocations of both `startInstance` and `saveIssuer`. We were unable to identify a way for either of the latter functions to return successfully if one of the failures were to occur.

**Depositing Hangs or Fails**
We investigated the security property that if a deposit fails or hangs while a user is making an *offer* (secured declaration of digital assets available to trade), then the offer does not succeed and any funds that were successfully deposited are no longer accessible to the user. Our team examined the code in `escrowStorage.js` and `offer.js` and attempted to force an offer call to succeed when a deposit is either rejected or has failed to fulfill. Likewise, we attempted to recover successfully deposited funds in these conditions and could not find any viable method for recovery, given that a seat is not returned by `offer` when a deposit fails.

**Bad Payout Payments**
We investigated payouts (which contain multiple payments), including the case when a payment fails due to a withdrawal failure associated with a misbehaving issuer. We reviewed `withdrawPayments` in `escrowStorage.js` and tried to attack the `brandToPurse` map from a misbehaving issuer. We did not identify a way in which misbehaving issuers would affect the other payments in the payout. We found that only the withdrawals for the misbehaving issuers would fail, resulting in loss of those parts of the payout. This is the intended behavior, as assets of a misbehaving issuer should have no value to the receiver.

## Testing & Experimentation

To further test the security properties of issuers, and the implications of misbehaving issuers on the security of the system, our team attempted to implement a mechanism to copy, mutate, and harden an object. We mutated issuer behaviour such that `burn()` does nothing or `burn()` throws, and observed what properties hold for these misbehaving issuers. Most parts of the implementation assert the frozen status of their inputs, in addition to further assertions, which makes it more difficult to pass those mutated values as trusted inputs.

We found that attempting to mutate a hardened object proved too difficult. The approach of copying an already valid, frozen object and changing it to introduce an issue was attempted with some Zoe functions. However, in our attempt to do so, we learned that there are some cases where the modified copies were not rejected (e.g. `addToAllocation` in a ZCF seat, which did not reject our modified input). As a mitigation, we suggest adding protections for passing amounts similar to those added by `checkRemotable`. However, this is out of scope for this assessment.

Amounts have a brand, and each brand has an issuer, allowing a tampered issuer to be trusted, if the issuer is retrieved from the amount. We also unsuccessfully attempted to force an issuer to pass an amount that has been tampered with.

In addition, we performed property based tests on `AmountMath` and found that the algebraic properties hold with no violations reports. In reviewing the `AmountMath` code, we realized that set addition and subtraction are not exactly set operations. In `AmountMath`, the addition operation is defined if A and B have zero elements in their intersection. Conversely, subtraction requires that B is a subset of A. This provides an interesting property, which is that, aside from identity addition, the order of additions and subtractions is important, unlike regular natural number addition.

The type for `AssetKind` may be expanded to allow higher-order specifications of assets (e.g. a set where all objects must have the shape `{ foo : { bar: null } }`, or a set where values can be one of a known list of valid values). This would allow for extra protections in `AmountMath` to ensure the asset kind does not change as a result of arithmetic operations.

## Code Quality

The ERTP and Zoe codebases are well organized into small, concise files that adhere to a meaningful naming convention. The code follows best practices for choosing long, clear variable names (e.g. `zcfSeatToStagedAllocations`) and there are many assertion statements throughout the code to check for correct behavior.

However, the code is dense, highly refactored, and uses many custom libraries rather than common standards (e.g. Notifiers instead of Observables). As a result, our team found that becoming familiar with the coding style and conventions (in addition to the overall system design) was particularly challenging. In many cases, functions relevant to a module are often defined in several other locations, requiring the reader to track them down. While the Agoric team reasonably demonstrated the necessity of these design choices, they do increase the amount of time and effort required to become familiar with the operation of the code and provide meaningful analysis.

### Tests

Sufficient test coverage has been implemented for both ERTP and Zoe, including unit tests that help reviewers and Zoe smart contract developers understand the intended functionality of each of the components, and check that they behave as intended.

## Documentation

The Agoric team provided accurate and comprehensive documentation, including a high level description of the system, each of the components, and interactions between those components. In addition to the documentation, the Agoric team provided sample smart contracts demonstrating the use of Zoe for common use cases.

The Agoric system presents several new ways of thinking and writing about smart contracts. We encourage the Agoric team to continue providing educational resources for beginners (e.g. a smart contract playground and tutorial series).

### Code Comments

The documentation contained within the code is comprehensive and explains in detail the intended functionality of each of the components.

## Scope and Dependencies

The Agoric system consists of many layers, with each layer performing a critical function. This includes cryptography, consensus, networking, and others, to create a platform in which a smart contract developer can work with ERTP and Zoe only at the level of JavaScript objects. For this review, our team examined the ERTP and Zoe layers, with the assumption that the rest of the layers that make up the Agoric system function as intended and do not introduce vulnerabilities into the ERTP and Zoe components.

We commend the Agoric team's strong considerations for security, as demonstrated by the ERTP and Zoe system design and corresponding implementations. Given that these are multifaceted and complex systems, with large and comprehensive codebases, we recommend that the Agoric team continue to consider various approaches to security due diligence. This may include, but is not limited to, adherence to widely accepted security best practices, continuous reviews by independent security auditing teams, and the creation of a bug bounty program to identify implementation errors that could be easily missed by a security review that is limited in scope.

## Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

| ISSUE / SUGGESTION | STATUS |
|---|---|
| Issue A: ERTP Trusted User Inputs | Resolved |
| Issue B: Possible Error Shadowing in Case of External Code Throwing Exceptions | Resolved |
| Issue C: Zoe Inconsistent States | Unresolved |
| Suggestion 1: Adhere to Keyword Standards | Resolved |
| Suggestion 2: Improve Behavior When Creating a Purse Hangs or Fails | Resolved |

## Issue A: ERTP Trusted User Inputs

**Location**

[ERTP/src/paymentLedger.js#L129](ERTP/src/paymentLedger.js#L129)

**Synopsis**

ERTP's internal `reallocate` function takes an array as input and calls `Array.forEach` and other standard array methods on it. However, objects passed in by the user should not be trusted. An object with the correctly named methods, but incorrect behavior, could be passed.

**Impact**

The `reallocate` function is not exposed publicly and the methods that do call it happen to copy the array. However, given access to this function, an attacker could create free money resulting in a loss of legitimate funds.

**Preconditions**

An attacker gains access to the `reallocate` function, or a method that calls `reallocate` uses the user-supplied payments array instead of copying it.

**Technical Details**

The methods that call `reallocate` use `Promise.all` on their input array. This uses the iterator interface to copy the array and resolves a new and safe array. However, if a user input were to be passed directly to `reallocate` and the `forEach` function misbehaved, it would be possible to duplicate payment objects. The `reallocate` function iterates over the payments and calculates the sum of their amounts. It then creates new payment objects with the same total amount, then calls `forEach` again to delete the original payment objects. If the `payment.forEach` method just did nothing on the second call, the payments would not be deleted, and they could be used to create new payment objects again. This would allow the caller to create free money.

`zcfSeat` also assumes that `forEach` behaves correctly, but is called by methods that copy the array.

**Remediation**

We recommend importing trusted methods and passing user objects to them, instead of trusting functions on objects passed in by users. For inputs such as arrays, we suggest copying the data to new trustworthy arrays. A simple way to copy the array is to use a destructuring pattern combined with the spread operator `` `([...array])`, `` which creates a new array that is a copy of the array elements.

We also recommend that the `reallocate` function perform this check on its `payments` argument because of the importance of the function.

**Status**

The Agoric team has [implemented](implemented) validations in `reallocate` (renamed to `moveAssets`) to check that the arrays passed to the function are `copyArrays`. This confirms that the values passed to the function are not proxy objects and that they do not implement any custom accessors.

**Verification**

Resolved.

## Issue B: Possible Error Shadowing in Case of External Code Throwing Exceptions

**Location**

`ERTP/src/paymentLedger.js#L34`

**Synopsis**

If `optShutdownWithFailure` fails and throws an error, it will prevent the supplied reason `shutdownLedgerWithFailure` from surfacing. Thus, `shutdownLedgerWithFailure` will exit prematurely before finishing execution and the original logical error (supplied by the smart contract) will be shadowed.

**Impact**

Errors supplied by the smart contracts for mitigating logical workflows will be shadowed by errors coming from `optShutdownWithFailure` instead of being propagated.

**Preconditions**

An issuer has implemented an optional shutdown with failure function, which throws its own exception.

**Remediation**

We recommend running `optShutdownWithFailure` within a `try-catch` block.

**Status**

The Agoric team has modified the `optShutdownWithFailure` usage to capture any exception thrown from the function, in addition to logging the original cause of failure instead of ignoring it.

**Verification**

Resolved.

## Issue C: Zoe Inconsistent States

**Location**

zoe/src/contractFacet/zcfZygote.js#L172

zoe/src/contractFacet/zcfZygote.js#L189

**Synopsis**

Zoe's ordering of state changes is not deterministic for each smart contract call, which leads to inconsistent intermediate states due to the ordering of messages across different vats.

**Impact**

Smart contract code will possibly have an inconsistent view due to the lack of ordering of asynchronous calls in cases with multiple remotables. In the referenced example, the Zoe smart contract will see a user allocation that is not escrowed by Zoe.

**Technical Details**

Asynchronous calls to different remotables have no guarantee on the execution order, which can lead to Zoe being in an unintended inconsistent state, particularly in the event that one of these asynchronous calls eventually fails.

The side-effectful messages sent to different remotables will lead to multiple possible states for the Zoe smart contract. However, if the side effects are ordered, then the ordering of messages across vats is guaranteed and the states follow a strict ordering.

**Mitigation**

We recommend chaining promises which update state on different remotables with `.then()`.

**Status**

The Agoric team has provided a high level overview of the intended change, which aligns with the suggested mitigation. Instead of chaining the promises using `then()`, the proposed solution aims to introduce a new function, `after()`, which would improve the pipelining behavior. The changes have not been implemented at the time of this verification.

**Verification**

Unresolved.

## Suggestions

### Suggestion 1: Adhere to Keyword Standards

**Location**

`zoe/src/contractFacet/zcfZygote.js#L160`

`zoe/src/contractFacet/zcfZygote.js#L178`

**Synopsis**

Keywords follow a standard that is being asserted and enforced to prevent accidental conflicts with existing property names in some cases (i.e. `instanceRecordStorage.js`, `cleanProposal.js`) but not in all cases, as indicated in the aforementioned links in the location section.

**Mitigation**

We recommend making all keywords adhere to the same standard.

**Status**

The Agoric team has exposed and modified `coerceAmountKeywordRecord`. The function has been hardened and asserts that all properties in a record are keywords. This function has been used in the locations where keywords were expected but were not checked.

**Verification**

Resolved.

## Suggestion 2: Improve Behavior When Creating a Purse Hangs or Fails

**Location**

`zoe/src/zoeService/escrowStorage.js#L22-L27`

`zoe/test/unitTests/test-zoe.js#L192-L204`

**Synopsis**

`startInstance` and `saveIssuer` create an empty purse as part of their implementation. If the empty purse creation fails, then both of these functions should fail.

The `createPurse` function calls the issuer's `makeEmptyPurse` method:

```
  const createPurse = (issuer, brand) => {
    if (!brandToPurse.has(brand)) {
      brandToPurse.init(brand, E(issuer).makeEmptyPurse());
    }
  };
```

If `makeEmptyPurse` fails, then a rejected promise will be stored in `brandToPurse` but the `createPurse` function will not fail. As a result, neither `startInstance` nor `saveIssuer` will fail as they should. We have created a unit test to demonstrate this problem.

**Mitigation**

`createPurse` is a synchronous function therefore it cannot return a rejected promise without changing its signature. We recommend rearchitecting the function and its callers so that failures in `makeEmptyPurse` bubble up the call stack.

**Status**

The Agoric team has implemented a change such that the call to `createPurse` blocks in the caller, which would cause the caller to throw or hang if `createPurse` throws or hangs.

**Verification**

Resolved.

## About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

## Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

### Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

### Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

## Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

## Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

## Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.