Succinct Labs Prover Network

Security Assessment

July 24, 2025

Prepared for:
John Guibas
Succinct Labs

Prepared by: **Tarun Bansal, Priyanka Bose, Marc Ilunga, Kevin Valerio**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Tara Goodwin-Ruffus, Project Manager
tara.goodwin-ruffus@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Kevin Valerio, Consultant
kevin.valerio@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 20, 2025	Pre-project kickoff call
July 8, 2025	Status update meeting #1
July 14, 2025	Delivery of report draft
July 14, 2025	Report readout meeting
July 24, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Succinct Labs engaged Trail of Bits to review the security of the Succinct Prover Network smart contracts and the `vapp` Rust crate. The Prover Network creates a two-sided marketplace between provers and requesters, enabling anyone to receive proofs for applications such as blockchains, bridges, oracles, AI agents, video games, and more. The smart contracts are used to manage provers, stakers, and requester deposits, and the `vapp` crate implements a state transition function to execute and prove user actions in the network.

A team of four consultants conducted the review from June 23 to July 11, 2025, for a total of seven engineer-weeks of effort. Our testing efforts focused on finding issues impacting the availability and integrity of the target system. With full access to source code and documentation, we performed static testing of the smart contracts and Rust codebase, using automated and manual processes. The auctioneer, sequencer, RPC, and other off-chain components were not in scope for this review.

Observations and Impact

The Succinct Prover Network smart contracts are divided into well-defined components with limited responsibility. They are also broken into small functions for readability and easy testability. The Rust `vapp` crate is also broken into multiple modules to manage the complexity and testability of the codebase. Both the smart contracts and `vapp` crate implement access control checks and validation checks to ensure system correctness, liveness, and security. However, multiple components rely on outside services for several security-critical checks. For example, the RPC server is used to lock user funds while user requests are being processed by the system instead of locking funds in the `vapp` state (TOB-SUCC-2). Additionally, the system relies on the correct functioning of the auctioneer service to clear the proof requests to ensure that the provers are paid for every valid proof generation. The dependence on outside services for critical checks can introduce security vulnerabilities with future upgrades of these services.

During the review, we discovered an issue of undetermined severity that causes user funds to be locked in the `SuccinctVApp` contract (TOB-SUCC-9), a medium-severity issue arising from a lack of a validation check (TOB-SUCC-10), a truncated cryptographic hash collision issue (TOB-SUCC-4), one front-running issue (TOB-SUCC-5), one denial-of-service vulnerability (TOB-SUCC-7), and several other issues related to the lack of sufficient data and access control validation checks (TOB-SUCC-1, TOB-SUCC-6, and TOB-SUCC-8).

Recommendations

Based on the codebase maturity evaluation findings identified during the security review, Trail of Bits recommends that Succinct Labs take the following steps before production deployment of the Succinct Prover Network:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactoring that may occur when addressing other recommendations.
- **Document the system state specification.** Create system state specification documentation with state transition diagrams, state transition preconditions, and descriptions of validation checks. Develop end-to-end test cases considering all possible actors and their interactions with the system. Refer to this document for implementing data and access control validation checks for every function.
- **Document security assumptions.** Document security assumptions of the current system and the outside services responsible for security checks. Refer to this document to guide the development of the outside services and ensure that every system upgrade validates these assumptions.
- **Redesign the reward distribution and incentive mechanisms.** Redesign the staker reward distribution mechanism to consider the staking period for the reward period. Consider malicious actors and buggy systems while designing and implementing incentive mechanisms. Evaluate risks arising from collusion among parties.
- **Perform a security audit of off-chain components.** Consider performing a security audit of the supporting off-chain components, such as the auctioneer service, sequencer, RPC server, verifier service, and execution server. These components implement several cryptographic security checks that should be reviewed for soundness to ensure end-to-end security of the Prover Network.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	3
Informational	5
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Authentication	1
Cryptography	1
Data Validation	7

Project Goals

The engagement was scoped to provide a security assessment of the Succinct Labs Prover Network. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the slashing mechanism properly implemented to penalize malicious provers while protecting honest participants?
- Can malicious actors exploit the unstaking process to bypass cooldowns or delays?
- Are there issues in the reward distribution mechanism that would allow malicious stakers to earn disproportionate rewards by staking for minimal durations?
- Can token transfers and approvals be exploited to perform unauthorized operations?
- Can user deposits and withdrawals be protected against theft or unauthorized access?
- Does the `step` function properly verify zero-knowledge (ZK) proofs for state transitions?
- Are critical protocol functions vulnerable to denial-of-service or out-of-gas attacks?
- Is the contract upgradeability mechanism properly restricted to authorized parties?
- Does the protocol use insecure governance parameters?
- Are off-chain transactions protected against replay and malicious signature attacks?
- Does the storage implementation maintain cryptographic integrity through Merkle proof validation?
- Are the state transition and aggregation circuits mathematically sound?
- Can a malicious prover bypass the whitelist controls set by a proof requestor?
- Is the fee distribution mechanism correctly implemented for protocol participants?
- Can provers submit invalid proofs while still collecting generation fees?
- Is the protocol resilient against Ethereum mainnet reorganizations?
- Are there vulnerabilities that could trigger panic conditions in on-chain transactions?

Project Targets

The engagement involved reviewing and testing the following target.

Succinct Labs Prover Network

Repository	https://github.com/succinctlabs/network
Version	`05ed9e3c60466ab92d8970ed6d5b4288d14f74b9`
Directories	`contracts`, `crates/vapp`
Type	Solidity, Rust
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Static analysis:** We analyzed the codebase using automated tools such as Slither and Clippy to identify common security vulnerabilities and code quality issues.
- **Manual code review:** We performed a detailed manual review of the codebase, focusing on the critical components, including the staking contracts, governor contracts, `vapp` contract, token contracts, and `vapp` crate.

Smart Contracts

- **`SuccinctStaking`:** The `SuccinctStaking` contract forms the foundation of the protocol's security model by managing staking and delegation mechanisms. This contract enables users to stake tokens, delegate their stakes to provers, and participate in protocol governance, while also implementing the critical slashing functionality that enforces honest behavior. During our manual review process, we focused on the following aspects:
    - **Initialization and access controls:** We verified the correctness of the contract configuration initialization and evaluated the implementation of owner privileges. We evaluated the implementation and application of the `onlyProver` and `stakingOperation` modifiers to check if an attacker can execute malicious transactions. We also looked for potential denial-of-service vulnerabilities in the staking, unstaking, slashing, and dispense functionalities.
    - **Slashing mechanism:** We reviewed the slashing functionality to ensure the system's integrity, verifying that honest provers cannot be unfairly penalized while malicious actors cannot legitimately evade penalization. Specifically, we looked for issues related to the accuracy of the slashing amount calculation and the possibility of front-running attacks between slashing and unstaking operations.
    - **Unstaking and rewards:** We analyzed the unstaking process for security vulnerabilities, looking for potential opportunities to bypass cooldowns and for flash loan exploits enabling same-block stake/unstake operations; we also assessed the balance calculation's accuracy. Our review also carefully assessed the reward distribution logic, focusing on the reward calculation and on identifying front-running attack scenarios between reward distribution and staking transactions.

- **Delegation and token handling:** We examined the delegation logic to verify that stakers cannot delegate to multiple provers simultaneously. Additionally, we evaluated the possibility of exploiting the existing smart contract token transfer approvals and validated that iProve tokens cannot be transferred to unauthorized non-Prover-N contracts. We also verified that voting unit transfers and transfers of iProve tokens work correctly. Finally, we also validated that each new prover contract implementation is properly deployed and initialized.

- **`SuccinctVApp`:** The `SuccinctVApp` contract serves as the core component managing the application's state and transactions on-chain. Our manual review focused on the following aspects of this contract:
  - **Initialization and access controls:** We verified the correctness of genesis block root initialization and evaluated the implementation of owner privileges and pause functionality throughout the system.
  - **Deposit and withdrawal functionality:** We conducted a comprehensive analysis of the deposit and withdrawal mechanisms to ensure the system protects user funds against theft and unauthorized access. We also investigated the possibility that the funds could become stuck in the contract. Our investigation looked for any potential weakness in the prover registration processes that could allow unintended registrations. Additionally, we rigorously verified that the `step` function properly and completely validates ZK proofs of state transitions, ensuring the integrity of the system's cryptographic guarantees.
  - **System-wide attack vectors:** Our evaluation looked for potential attack vectors, including denial-of-service and out-of-gas vulnerabilities on critical functions, particularly in the `step` function while processing large receipt volumes. In addition, we analyzed the correctness of the fork function, contract upgradeability, and receipt validation in the `Receipts` library.

- **`SuccinctGovernor`:** The `SuccinctGovernor` contract implements the protocol's governance mechanisms. With most of the parts inherited from OpenZeppelin's governance library, we focused our review on verifying the correct configuration of all the governance parameters and thoroughly assessed the inheritance structure to ensure the intended behavior of the overridden functions.

- **Token and ERC-4626 vault contracts:** The Prover Network implements PROVE and stPROVE ERC-20 tokens, and iPROVE and PROVER-N ERC-4626 tokenized vaults. We checked the ERC-20 tokens for the correct implementation of the ERC-20 standard and access control checks on the mint, burn, approval, and transfer functions. The `SuccinctProver` contract implements an ERC-4626 tokenized vault that accepts iPROVE tokens and mints non-transferable PROVER-N tokens for delegating stake to

provers. With most functionality inherited from OpenZeppelin's ERC-4626 implementation, we focused our review on identifying vulnerabilities specific to tokenized vaults, such as exchange rate manipulation, share price inflation attacks, donation-related vulnerabilities, first depositor exploitation, and rounding issues. We also assessed the system's compatibility with nonstandard tokens, decimal precision handling, and compliance with the ERC-4626 specification's requirements for preview functions and conversion mechanics.

Vapp Crate

The `vapp` crate implements the core off-chain logic for the application protocol, handling state transitions, transaction processing, and proof generation. This critical component manages the protocol's cryptographic verification systems and ensures secure interaction between on-chain and off-chain operations. Our primary analysis of the `vapp` crate consisted of identifying any unsafe arithmetic operations and dependency on any vulnerable third-party libraries. We measured the crate's test coverage using tarpaulin, used Clippy for linting rules, used `cargo-audit` to identify vulnerable dependencies, and used `cargo-updeps` to find unused crates. In addition to that, our review focused on identifying various attack scenarios in the following different functionalities:

- **Storage and state management:** We evaluated the integrity and security of the `vapp` crate's storage that maintains application state. During the analysis, we looked for issues related to storage bloating that could impact performance or enable resource exhaustion, cryptographic weaknesses in the storage implementation that might lead to invalid Merkle proof validation, and vulnerabilities to storage key collision attacks that could corrupt application state data; we also checked the correctness and soundness of storage recovery and update functions.
- **Transaction processing:** This `vapp` functionality manages off-chain transaction processing for various transaction types such as deposits, withdrawals, and transfers. While reviewing this functionality, we assessed the signature verification logic for completeness and correctness, and we looked for potential vulnerabilities that could bypass atomic transaction properties or reversibility constraints and for replay attack vectors where captured signatures might be reused for unauthorized operations. We also evaluated the effectiveness of access control implementations across all critical transaction types to ensure proper permission enforcement throughout the system.
- **Proof generation and verification:** The `vapp` crate's cryptographic proof system uses ZK proof verification to validate state transitions between on-chain and off-chain components. We looked for potential security vulnerabilities, including whether requesters could bypass punishment penalties when submitting unexecutable programs and if provers could circumvent requester-defined whitelists. We rigorously evaluated the mathematical soundness of state transition circuits and their input validation mechanisms. Additionally, we investigated

whether provers could exploit the system by submitting fulfillment bodies without valid cryptographic proofs while still collecting generation fees, an attack vector that would fundamentally undermine the protocol's economic security model.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Off-chain components:** Off-chain components, including the auctioneer service, sequencer, RPC server, execution service, verifier server, and prover server, were out of scope and were not reviewed during this engagement.
- **The `SP1Verifer` contracts:** The `SP1Verifer` contracts were out of scope and were not reviewed.
- **Deployment scripts:** The deployment scripts were out of scope and were not reviewed.
- **The Ethereum blockchain reorganization risks:** The Ethereum blockchain reorganization risks in both the smart contracts and `vapp` state transition function were not analyzed, assuming that the auctioneer service waits for Ethereum block finalization before processing Ethereum transactions in the `vapp` crate.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use open-source static analysis, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	No explicit policy, as the rules created run for under a few seconds

Areas of Focus

Our automated testing and verification work focused on the following:

- Sending of arbitrary ERC-20 tokens
- Reentrancy attack vectors
- Tokens becoming locked
- Unchecked low-level calls

Test Results

The results of this focused testing are detailed below.

Property	Tool	Result
Sending of arbitrary ERC-20 tokens	Slither	**Pass**
Reentrancy attack vectors	Slither	**Pass**
Tokens becoming locked	Slither	**Pass**
Unchecked low-level calls	Slither	**Pass**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The vapp crate performs arithmetic operations on 256-bit integers for balance calculations, transfers, and fee distributions. The codebase uses checked and saturating operations and error handling for most of the calculations. However, we identified inconsistencies in the arithmetic operations throughout the codebase. While some calculations employ safe arithmetic, others use raw arithmetic operations, such as adding values to constants. One identified issue relates to this inconsistency, combined with the use of an unchecked arithmetic operation (TOB-SUCC-3). The smart contracts use arithmetic operations only for the ERC-4626 vaults and staking reward calculations. The ERC-4626 vault calculations are inherited from the OpenZeppelin library. The staking reward calculations are simple and use the default rounding direction for the division operation. However, all arithmetic operations in the codebase can benefit from a systematic rounding direction strategy in favor of the protocol.	**Moderate**
Auditing	Most of the critical operations within the vapp crate emit logs via the tracing crate for each off-chain transaction for monitoring and tracking purposes. However, it is unclear whether those logs are stored on disk. We recommend creating a backup system for the logs to avoid scenarios where the team would need to investigate past issues or incidents without access to historical data. Smart contract events in the Succinct Prover Network play an important role since they ensure proper	**Satisfactory**

	communication between on-chain and off-chain components. The smart contracts emit events from all the state-changing functions to notify off-chain components and monitoring systems.	
Authentication / Access Controls	The Prover Network relies on cryptographic security and validation checks for authentication and access control checks. The system splits the cryptographic checks between on-chain and off-chain components without compromising on security. However, we identified several issues related to insufficient checks (TOB-SUCC-1, TOB-SUCC-4, TOB-SUCC-6, TOB-SUCC-8). Implementing end-to-end test cases that consider malicious behavior can help secure the system against such issues. Additional documentation specifying the roles, their privileges, and the process of granting and revoking the roles can also benefit the system.	**Moderate**
Complexity Management	In general, the code is well structured with small modules and functions that are easy to test. Each function has a specific and clear purpose and is clearly documented. The codebase shows strong error handling and type safety, with many variants for error handling and the use of strong rather than primitive types. Code duplication is limited, and the naming convention is strictly followed. However, the core transaction processing logic is implemented in a single function, which handles six different transaction types through a large match statement with deep nesting levels. This logic should be reorganized and maintained across various functions to improve the readability, maintainability, and testability of the transaction processing logic.	**Moderate**
Cryptography and Key Management	The codebase uses safe cryptographic primitives for signing data. The Merkle tree implementation uses a fixed depth to prevent the most common issues related to invalid Merkle proof validation. However, we identified an issue related to index collision risk. The codebase can benefit from explicit index validation and edge test cases for extreme scenarios.	**Satisfactory**
Decentralization	The Succinct Prover Network does not keep custody of user funds in a centralized wallet. Privileged actors are	**Weak**

	not able to unilaterally move funds out of, or trap funds in, the protocol. The smart contracts implement a governance system for updating crucial configuration parameters. However, the smart contracts are upgradeable by a single owner or a multisig wallet. Implementing timelock functionality for smart contract upgrades and configuration changes can improve the system's decentralization by allowing users time to opt out of such changes. Additionally, the off-chain systems, such as the auctioneer service and RPC node, use a centralized database for storing transaction data. These off-chain components are a single point of failure for the Prover Network. Similarly, the risks related to trusted third-party components and their behavior are not clearly documented.	
Documentation	The documentation includes comprehensive Markdown files with specifications for the protocol, inline comments, and external documentation on the Succinct Docs website that includes basic diagrams and user stories. Consistent naming conventions are followed throughout the codebase. The Rust functions' arguments and returned values are commented properly, though the documentation at the beginning of each Rust file is minimal. It typically consists of single-sentence descriptions that could be expanded to provide better context about each module's role within the protocol. The smart contracts include extensive inline comments on both the contracts and functions. The documentation can be improved to include system state specification, expected system behavior, and system invariants.	**Satisfactory**
Low-Level Manipulation	The smart contracts do not implement assembly code blocks. The `vapp` crate does not use any `unsafe` blocks.	**Not Applicable**
Testing and Verification	The test suite includes comprehensive unit and integration tests with strong coverage for most components, achieving high coverage across the `vapp` logic. However, the tests rely heavily on mocked environments that do not accurately and fully reflect the	**Moderate**

	complexity of real-world interactions, especially between on-chain and off-chain components. Additionally, there are no end-to-end tests validating the complete `vapp` workflow from user transactions through the auctioneer service, SP1 proof generation, and final on-chain settlement. We recommend replacing simplified mocks with more realistic tests that better simulate real-life behaviors, and implementing end-to-end tests that exercise the full `vapp` stack.	
Transaction Ordering	The Prover Network enforces a time delay for crucial user actions, such as staking, unstaking, and slashing, to protect users from front-running risks. However, we identified a front-running issue in the reward distribution function (TOB-SUCC-5). The `vapp` crate does not implement a fee market for off-chain transactions and relies on the sequencer to order transactions correctly. Finally, the documentation and tests lack a focus on front-running risks.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Insufficient signer checks for off-chain withdrawals allow unauthorized transactions	Access Controls	Low
2	ZK proof requesters can avoid punishment for unexecutable programs by maintaining a low balance	Data Validation	Informational
3	Unchecked arithmetic operation in the add_balance function allows integer overflow	Data Validation	Informational
4	Truncated index for RequestID can lead to collisions in storage keys	Cryptography	Informational
5	Stakers can front run the dispense transaction to earn rewards	Data Validation	Low
6	Missing deadline validation allows replay of expired proof requests	Data Validation	Informational
7	Unbounded receipt processing in _handleReceipts can cause denial of service	Data Validation	Low
8	Insufficient transaction validation in _handleOnchainReceipt could lead to processing of invalid transactions	Data Validation	Informational
9	The SuccinctVApp contract can lock the funds deposited by smart contracts	Authentication	Undetermined
10	Lack of fulfill_body verification for Compressed proofs	Data Validation	Medium

Detailed Findings

1. Insufficient signer checks for off-chain withdrawals allow unauthorized transactions

Severity: **Low**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-SUCC-1
Target: `crates/vapp/src/state.rs`	

Description
An insufficient authorization check in the `vapp` crate's `Withdraw` transaction allows anyone to trigger a withdrawal for a prover account.

The authorization checks for off-chain withdrawals validate the transaction signer only for the non-prover accounts. The transaction signer check is bypassed entirely for accounts with nonzero owner addresses, as they are prover accounts. As a result, any user can trigger a withdrawal for any prover account.

An attacker can repeatedly execute withdrawal transactions to grief a target prover account. The prover will not be able to execute off-chain transfers anymore, due to a lack of funds. However, the attacker cannot steal the funds, as the funds are always transferred to the correct prover when the withdrawals are processed in the smart contracts.

```
// If the account is not a prover (provers always have a non-zero owner address),
// then only the account itself can withdraw.
if owner == Address::ZERO && account != from {
    return Err(VAppPanic::OnlyAccountCanWithdraw);
}
```

Figure 1.1: The transaction signer validation of the withdrawal transaction (`crates/vapp/src/state.rs#L454–L458`)

Exploit Scenario
Eve identifies Alice's prover account (nonzero owner address) with 1,000 PROVE tokens. Eve creates a withdrawal transaction targeting Alice's account, signing it with her own key. Since the authorization check applies only to non-prover accounts, Eve's transaction passes validation. The system withdraws funds from Alice's account (though they go to Alice, not Eve). Eve repeats this attack every time Alice's balance increases, preventing Alice from making any off-chain transfers due to insufficient funds.

Recommendations

Short term, add a check to allow only the prover owner to sign the `Withdraw` transactions for the prover account.

Long term, expand the test suite to test the `OnlyAccountCanWithdraw` error and implement authorization tests to verify all access control checks.

2. ZK proof requesters can avoid punishment for unexecutable programs by maintaining a low balance

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SUCC-2
Target: `crates/vapp/src/state.rs`	

Description

The executor, an off-chain service, validates proof requests before provers process them, executing programs to gather metadata and determining whether requests can proceed to fulfillment. When the executor marks a proof request program as `Unexecutable`, the system attempts to deduct a punishment amount from the requester's balance to avoid cases where the requester is malicious and does not provide a well-formed request that can actually be proven, preventing spam requests. However, if the requester lacks sufficient balance, the `Clear` transaction fails with an `InsufficientBalance` error, and the requester is not punished for submitting an unexecutable request program.

```
// Validate that the requester has sufficient balance to pay the punishment.
let balance = self.accounts.entry(request_signer)?.or_default().get_balance();
if balance < punishment {
    return Err(VAppPanic::InsufficientBalance {
        account: request_signer,
        amount: punishment,
        balance,
    });
}
```

Figure 2.1: A snippet of the `Clear` transaction handler (`crates/vapp/src/state.rs#L643–L651`)

The client provided the following context for the issue:

> *"The RPC does validate that Alice has enough balance to pay for all its proofs. It deducts the maximum cost of a proof per request and then refunds the balance once the request gets fulfilled. The RPC, in general, maintains a postgres database with its own representation of the state of the system."*

Exploit Scenario

Eve submits an unexecutable proof request with high computational cost. The executor determines that the request is unexecutable and assigns a punishment. However, Eve transfers her balance to another account with an off-chain transfer to avoid punishment. The `Clear` transaction fails because of insufficient balance without punishing Eve. Eve

spams the network and potentially creates a denial-of-service issue by submitting multiple unexecutable requests.

Recommendations

Short term, lock funds in the `vapp` database when proof requests are submitted to prevent off-chain transfers before settlement.

Long term, consider malicious behavior while designing incentive mechanisms and system features. Implement end-to-end integration tests for all the system components to identify issues arising from the integration of multiple systems and features.

3. Unchecked arithmetic operation in the add_balance function allows integer overflow

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-3
Target: `crates/vapp/src/state.rs`	

Description

The `Account` struct represents users within the `vapp` state. The `add_balance` function of the `Account` struct is used to increase the associated account balance when processing deposits and transfers and when distributing proof generation fees.

However, the function's arithmetic logic is vulnerable to integer overflow. Since the `+` operator instead of a safe function like `checked_add` is used to increase the balance, a large amount could cause the account balance to overflow and decrease, leading to an inconsistent state. Such an overflow would require an account to have a balance that is approaching `U256::MAX`, which is almost unrealistic, making this issue difficult to trigger.

```
/// Adds an amount to the balance of the account.
pub fn add_balance(&mut self, amount: U256) {
    self.balance += amount;
}
```

Figure 3.1: The `add_balance` function of the `Account` struct (`crates/vapp/src/sol.rs#L130–L132`)

Additionally, some functions use saturating arithmetic operations. These operations silently ignore integer overflow issues by setting the maximum or minimum value as the arithmetic operation result. Such operations can also lead to inconsistent states and incorrect account balances.

Exploit Scenario

Eve, a requester, realizes that the treasury account contains an extremely large amount of PROVE tokens, nearly reaching the U256 maximum. Eve submits an unexecutable proof request, which triggers a punishment. The punishment amount is added to the treasury account, causing the balance to exceed the U256 maximum value. This results in an overflow that resets the treasury balance to a lower amount after wrapping the addition operation result around the U256 limit.

Recommendations

Short term, use the `checked_add` operation in the `add_balance` function. Replace the use of all saturating arithmetic operations with the checked arithmetic operations throughout the codebase to capture integer overflows.

Long term, ensure that native mathematical operators are never used within the codebase to avoid such issues. Adding fuzzing and static analysis tools such as Semgrep into the development pipeline can help to detect such bugs.

4. Truncated index for RequestID can lead to collisions in storage keys	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-SUCC-4
Target: `crates/vapp/src/storage.rs`	

Description

The vapp state keeps track of accounts and transactions in a Merkleized key-value store. The transaction store uses the request hash, also called `RequestID`, as a key for the storage. However, the request hash is truncated to 128 bits, increasing the risk for collisions.

Figure 5.1 shows how indexes are defined for the request store. The index is computed as a truncation of a `RequestID` object.

```
/// The unique identifier hash of a [`spn_network_types::RequestProofRequestBody`].
pub type RequestId = [u8; 32];

impl StorageKey for RequestId {
    fn index(&self) -> U256 {
        U256::from_be_slice(&self[..16])
    }

    fn bits() -> usize {
        128
    }
}
```

Figure 4.1: `RequestID` truncated to 128 bits (`crates/vapp/src/storage.rs#L66–L77`)

The Prover Network assumes that there will be around 2^{48} requests in the network. At this workload, two indices will collide with a probability of 2^{-32}. Although the latter probability is acceptable, it still represents a nonnegligible probability that valid requests will be rejected.

Recommendations

Short term, increase the size of the key in the transaction storage to 160 bits to ensure a practically negligible collision probability.

5. Stakers can front run the dispense transaction to earn rewards	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-5
Target: `contracts/src/SuccinctStaking.sol`	

Description

The `dispense` function of the `SuccinctStaking` contract distributes the rewards among all the stakers without considering a staker's staking period during the reward period. This allows any staker to front run the `dispense` transaction to add a large stake and unstake immediately after the reward distribution to capture a large portion of the reward without staking their funds for the full reward period. The other stakers who staked their funds for the whole reward period would lose out on their potential reward earnings to such just-in-time stakers. This undermines the staking incentive mechanism and allows opportunistic actors to extract value at the expense of long-term stakers.

Exploit Scenario

Eve notices a `dispense` transaction in the mempool that will distribute 100 PROVE in rewards. She quickly stakes 1,000 PROVE before the `dispense` transaction is processed, receives a proportional share of rewards, and then immediately unstakes. Eve has now received rewards without contributing to the security of the protocol through long-term staking.

Recommendations

Short term, implement a minimum staking period before accounts become eligible for reward distribution, or use a snapshot mechanism that records eligible stakers at a previous block.

Long term, redesign the reward distribution system to use time-weighted staking, where rewards are proportional to both stake amount and duration, disincentivizing short-term opportunistic staking behavior.

6. Missing deadline validation allows replay of expired proof requests

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-6
Target: `crates/vapp/src/state.rs`	

Description
The `vapp` crate does not validate the request deadline while processing `Clear` transactions, allowing expired proof requests to be fulfilled and settled on-chain.

The prover network implements a `Clear` transaction to finalize a proof request by verifying all signatures and transferring payment from the requester to the prover. A failed `vapp` transaction leads to a `VAppPanic` error, which causes the STF program to panic; as a result, the public values (the new state) are not committed. If an attempt to clear a proof request fails, an attacker can capture the transaction inputs, including the `request`, `bid`, `execute`, and `settle` bodies with their signatures, and replay the `Clear` transaction after the proof request deadline has passed.

The client informed us that the deadline is validated in the sequencer, and the intention of making transactions replayable is to correct sequencer ordering issues. However, the state transition function should be responsible for its own safety constraints rather than relying on external components.

Exploit Scenario
Alice submits a proof request with a one-hour deadline. Bob wins the auction and generates a valid proof, but the `Clear` transaction fails due to an insufficient balance error. The zkVM panics, and the state is not committed: the transaction is not inserted into the executed transaction list. Eve monitors the network and captures Bob's signed transaction data. One hour later, Alice tops up her balance to gain enough funds to cover the cost, but does not wish to reexecute her past transaction. Two hours later, well past the deadline, Eve replays the captured transaction. The `vapp` processes it successfully, and the expired request is fulfilled with Alice's payment.

Recommendations
Short term, add deadline validation within the `Clear` transaction handler to reject proof fulfillments that occur after the request deadline has passed.

Long term, add tests to ensure deadline validation. Consider how an attacker could exploit a feature to replay transactions to avoid this category of issue in the future.

7. Unbounded receipt processing in _handleReceipts can cause denial of service

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-7
Target: `contracts/src/SuccinctVApp.sol`	

Description

The `_handleReceipts` function implements an unbounded loop to process off-chain transaction receipts, which can lead to an out-of-gas exception and denial of service.

The `step` function of the `SuccinctVApp` contract calls the `_handleReceipts` function to process the receipts generated by the `vapp` crate state transition function while processing transactions. The zkVM processes multiple transactions and collects their receipts in an array, which are then processed by the `SuccinctVApp` smart contract after verifying the ZK proof generated by the zkVM for executing all the transactions.

As shown in figure 7.1, the `_handleReceipts` function in the `SuccinctVApp` contract processes all the receipts in the array in an unbounded `for` loop.

```
function _handleReceipts(StepPublicValues memory _publicValues) internal {
    // Execute the receipts.
    for (uint64 i = 0; i < _publicValues.receipts.length; i++) {
        if (_publicValues.receipts[i].onchainTxId != type(uint64).max) {
            _handleOnchainReceipt(_publicValues.receipts[i]);
        } else {
            _handleOffchainReceipt(_publicValues.receipts[i]);
        }
    }
}
```

Figure 7.1: Snippet of the `_handleReceipts` function (`contracts/src/SuccinctVApp.sol#L352-L361`)

The `vapp` crate and the `SuccinctVApp` contract do not limit the size of the receipts array being processed in a single transaction. Therefore, a sufficiently large array of receipts will inevitably cause transactions to run out of gas and fail. This occurs because the function iterates through all receipts in a single transaction without considering gas limitations.

Exploit Scenario

An attacker creates multiple small withdrawal requests in the `vapp` crate, each generating a receipt that must be processed. When these receipts are processed in a batch, the

`_handleReceipts` function exceeds the gas limit, causing the transaction to fail and preventing any receipts from being processed. Thus, withdrawals for all users are effectively blocked.

Recommendations

Short term, implement a maximum batch size for receipt processing and add a mechanism to handle receipts in multiple transactions when the total gas usage exceeds this limit.

Long term, redesign the receipt processing system to use a Merkle tree approach, where a receipt root is stored for each block and individual receipts can be processed with Merkle proofs. This will allow for bounded gas usage per transaction while maintaining cryptographic verification.

8. Insufficient transaction validation in _handleOnchainReceipt could lead to processing of invalid transactions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-8
Target: `contracts/src/SuccinctVApp.sol`	

Description
The `_handleOnchainReceipt` function does not explicitly validate the transaction state, potentially allowing invalid receipts to be processed.

The `_handleReceipts` function calls the `_handleOnchainReceipt` function to process the receipts generated by the `vapp` crate while processing the on-chain transactions. However, the `_handleOnchainReceipt` function in the `SuccinctVApp` contract lacks a verification step to explicitly check that the transactions being processed are in the `Pending` state before executing them. While the current implementation has implicit protections through receipt sequence validation and `assertEq` checks, additional explicit state validation would provide defense-in-depth against potential future vulnerabilities.

Exploit Scenario
A malicious actor submits receipts for processing with carefully crafted values that pass the current validation checks. Due to insufficient explicit validation, the system might process receipts for transactions that should not be eligible, potentially leading to unexpected system states or bypassing intended restrictions.

Recommendations
Short term, add explicit validation checks in the `_handleOnchainReceipt` function to verify that the transaction exists and is in a pending state before processing it, rather than relying on implicit validation through the equality check.

Long term, consider adding automated testing for edge cases involving transaction validation and documenting the expected validation workflow to prevent future code changes from introducing vulnerabilities.

9. The SuccinctVApp contract can lock the funds deposited by smart contracts

Severity: **Undetermined**	Difficulty: **Low**
Type: Authentication	Finding ID: TOB-SUCC-9
Target: `contracts/src/SuccinctVApp.sol`	

Description

The smart contracts depositing PROVE tokens to the `SuccinctVApp` contract cannot withdraw those tokens because the `Withdraw` off-chain transaction processor requires a signature from the account withdrawing the tokens.

The `SuccinctVApp` contract allows users, both EOAs and smart contracts, to deposit PROVE tokens to pay for their proof requests, as shown in figure 9.1. Users can withdraw their remaining tokens by sending a `Withdraw` transaction to the sequencer, which is processed by the state transition function, as shown in figure 9.2.

```
function deposit(uint256 _amount) external override whenNotPaused returns (uint64
receipt) {
    return _deposit(msg.sender, _amount);
}
```

Figure 9.1: The `deposit` function of the `SuccinctVApp` contract (network/contracts/src/SuccinctVApp.sol#L170–L172)

```
VAppTransaction::Withdraw(withdraw) => {
    [...]
    // Verify the proto signature.
    debug!("verify proto signature");
    let from = proto_verify(body, &withdraw.withdraw.signature)
        .map_err(|_| VAppPanic::InvalidWithdrawSignature)?;

    [...]
    // Extract the account address.
    debug!("extract account address");
    let account = address(body.account.as_slice())?;
    let owner = self.accounts.entry(account)?.or_default().get_owner();

    // If the account is not a prover (provers always have a non-zero owner
address),
    // then only the account itself can withdraw.
    if owner == Address::ZERO && account != from {
        return Err(VAppPanic::OnlyAccountCanWithdraw);
```

```
    }
    [...]
}
```

Figure 9.2: The `Withdraw` transaction handler of the state transition function (`audit-succinct-pn/crates/vapp/src/state.rs#L405–L499`)

The state transition function authenticates the `Withdraw` transaction by validating that the account being withdrawn signs the `WithdrawRequestBody`.

However, smart contracts cannot sign a `Withdrawal` transaction, so they cannot withdraw PROVE tokens they deposit into the `SuccinctVApp` contract. The funds deposited by smart contracts are stuck forever in the `SuccinctVApp` contract, resulting in a loss to the depositor.

Additionally, the same issue affects the off-chain `Transfer` transaction. If a user transfers their balance to a smart contract account using an off-chain `Transfer` transaction, then they cannot withdraw or transfer the tokens back to their own account, losing the transferred tokens forever.

Exploit Scenario

Alice, developer of a zkVM dapp, integrates her smart contract with the `SuccinctVApp` contract to deposit PROVE tokens to automate her proof request flow. The smart contract deposits 1 million PROVE tokens in advance to process proof requests for a month. Alice decides to withdraw 500,000 PROVE tokens after stopping her dapp. Alice cannot withdraw the PROVE tokens and loses a large amount of her initial investment.

Recommendations

Short term, restrict the `SuccinctVApp` contract's `deposit` function to EOAs only. Implement a check in the `Transfer` function handler to ensure that the token receiver is an EOA that can sign the off-chain requests.

Long term, create system state specification documentation with state transition diagrams, state transition preconditions, and validation checks. Develop end-to-end test cases considering all possible actors and their interactions with the system to identify and resolve such issues.

10. Lack of fulfill_body verification for Compressed proofs

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SUCC-10
Target: `crates/vapp/src/state.rs`	

Description

The `Clear` transaction handler does not validate the `fulfill_body` value to ensure it includes the correct proof. A malicious or malfunctioning auctioneer can submit invalid or empty proofs in the `Clear` transaction.

The Prover Network supports three types of proofs: `Compressed`, `Groth16`, and `Plonk`. The `Clear` transaction handler accepts the `request`, `bid`, `settle`, `execute`, and `fulfill` data, validates all of this data, and distributes the proof generation reward among the protocol, prover, and stakers. The `Clear` transaction handler validates the `fulfill_body` value to ensure it includes a valid proof by checking the `verifier` signature on the `fulfillment_id` for `Groth16` and `Plonk` proofs.

However, the transaction handler validates `Compressed` proofs by reading the proof data from the zkVM input stream instead of reading it from the `fulfill_body` value, as shown in figure 1:

```
let execute_public_values_hash: [u8; 32] = execute
    .public_values_hash
    .as_ref()
    .ok_or(VAppPanic::MissingPublicValuesHash)?
    .as_slice()
    .try_into()
    .map_err(|_| VAppPanic::FailedToParseBytes)?;
let public_values_hash: [u8; 32] = match &request.public_values_hash {
    Some(hash) => {
        let request_public_values_hash = hash
            .as_slice()
            .try_into()
            .map_err(|_| VAppPanic::FailedToParseBytes)?;
        if request_public_values_hash != execute_public_values_hash {
            return Err(VAppPanic::PublicValuesHashMismatch);
        }
        request_public_values_hash
    }
    None => execute_public_values_hash,
};
```

```
// Verify the proof.
let vk = bytes_to_words_be(
    &request
        .vk_hash
        .clone()
        .try_into()
        .map_err(|_| VAppPanic::FailedToParseBytes)?,
)?;
let mode = ProofMode::try_from(request.mode)
    .map_err(|_| VAppPanic::UnsupportedProofMode { mode: request.mode })?;
match mode {
    ProofMode::Compressed => {
        let verifier = V::default();
        verifier
            .verify(vk, public_values_hash)
            .map_err(|_| VAppPanic::InvalidProof)?;
    }
```

Figure 10.1: The `Compressed` proof verification (`audit-succinct-pn/crates/vapp/src/state.rs#L709–L746`)

Therefore, a malicious or malfunctioning auctioneer can submit a `fulfillment` with empty or invalid proofs in a `Clear` transaction to disrupt the components relying on the `Clear` transaction body to get proof data.

Note that a valid proof should be accessible to the auctioneer service for the proof request to successfully execute the `Clear` transaction, because it is verified.

Exploit Scenario

An upgrade to the auctioneer service introduces a bug that inserts empty proofs in the `fulfill_body` value of `Clear` transactions. The `Clear` transaction is executed successfully, but requesters cannot find the proof data by reading it.

Recommendations

Short term, add a check to validate that the `fulfill_signer` is associated with the assigned prover for `Compressed` proofs.

Long term, consider the malicious and malfunctioning of system components when designing security checks for the system.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Improve function naming for clarity.** The `root()` and `timestamp()` functions in `SuccinctVApp.sol` should be renamed to clearly indicate they return values for the latest block number, for instance, with `latestRoot()` and `latestTimestamp()` to make the intent explicit.
- **Remove outdated and incorrect comments.** This comment in `SuccinctVApp.sol` is outdated and no longer reflects the actual code behavior.
- **Eliminate code duplication across dependencies.** The `FulfillmentStrategy` enum is defined separately in both SP1 and network dependencies rather than being shared from a common source. This duplication creates maintenance risks where changes to one definition may not be reflected in the other.
- **Add explicit bounds validation for Merkle tree indices.** The `verify_proof` function does not validate that U256 keys are within the expected tree size bounds. While key types implicitly enforce range constraints, explicit validation would improve code clarity and prevent potential confusion.
- **Apply whitelist validation only where appropriate.** The `Clear` transaction handler currently validates prover whitelists for all fulfillment strategies, but according to the protocol documentation, prover whitelist validation should apply only to requests opting for the auction strategy.
- **Remove unused dependencies from the codebase.** Several Rust dependencies, including `lazy_static`, `alloy`, and `sha3`, are declared but not used in the project.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.