EVAA Finance

Security Assessment

August 22, 2025

Prepared for:
Vladislav Kamyshov
Something Labs LTD

Prepared by: **Guillermo Larregay, Quan Nguyen, and Kevin Valerio**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Quan Nguyen, Consultant
quan.nguyen@trailofbits.com

Kevin Valerio, Consultant
kevin.valerio@trailofbits.com

Project Timeline

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

Date	Event
May 29, 2025	Pre-project kickoff call
June 9, 2025	Status update meeting #1
June 16, 2025	Status update meeting #2
June 23, 2025	Status update meeting #3
June 27, 2025	Delivery of report draft
June 27, 2025	Report readout meeting
August 22, 2025	Delivery of final comprehensive report with fix review

Executive Summary

Engagement Overview

Something Labs LTD engaged Trail of Bits to review the security of EVAA Finance, a lending protocol built on the TON blockchain. Users of the system can deposit and withdraw collateral, become liquidity providers, and borrow and repay overcollateralized loans. The protocol supports several assets, including native TON coins and well-known Jettons.

A team of three consultants conducted the review from June 2 to June 27, 2025, for a total of eight engineer-weeks of effort. Our testing efforts focused on the different user interaction flows: supply assets, withdraw assets, liquidate unhealthy positions, and administrative operations: contract upgrades, system pause and restart procedures, and parameter changes. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

From July 16 to July 18, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the EVAA Finance team for the issues identified in this report. In summary, the team has resolved all but two informational issues. See appendix H for information on the two unresolved issues.

Observations and Impact

The EVAA protocol demonstrates significant complexity as a mature DeFi lending platform with extensive functionality spanning multiple contract types. While the codebase shows thoughtful engineering with modular design and comprehensive error handling, the complexity introduces substantial maintenance challenges and potential attack vectors.

We identified one high-severity issue related to insufficient fee validation in the liquidation process that could permanently lock user funds (TOB-EVAA-7). We discovered three medium-severity issues: one involving a faulty pause mechanism that fails to block certain operations (TOB-EVAA-1), one related to improper upgrade configuration persistence that could break future upgrades (TOB-EVAA-2), and one related to incorrect gas handling that could lock TON supply amounts (TOB-EVAA-3). We also found four informational issues related to incorrect rounding in borrow calculations (TOB-EVAA-4), timing issues with origination fee application (TOB-EVAA-5), supply cap bypass vulnerabilities through race conditions (TOB-EVAA-6), and the potential impact of liquidation threshold changes on existing positions (TOB-EVAA-8).

The protocol's centralized upgrade mechanism and administrative controls present significant trust assumptions, as the administrative account can unilaterally modify system parameters and force contract upgrades without user consent. While privileged accounts cannot directly access user funds, the broad administrative powers represent a meaningful centralization risk.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Something Labs LTD take the following steps:

- **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.
- **Improve the test suite.** The current test suite can be improved to eliminate existing coverage gaps. EVAA is a complex system with several internal and external interactions; having a robust test suite that covers the entirety of the codebase is important to detect issues early. Consider integrating tests into the CI/CD pipeline.
- **Update the documentation.** Current documentation available in the website lacks technical depth for advanced users or third-party developers. Include diagrams, algebraic expressions, and other low-level information that could help people interact with the protocol at different levels.
- **Consider refactoring the codebase.** The code can be complex to read and understand for new users and developers due to the amount of functions and variables passed around in calls and messages. A codebase that has no redundant variables or function parameters is cleaner, easier to read, and easier to maintain.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

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

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	8
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of the Something Labs LTD EVAA Finance codebase. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker manipulate the supply process to bypass supply constraints or steal user deposits?
- Can users withdraw more than their collateralized amount through manipulation of the supply-withdraw flow?
- Can the multistep supply-withdraw operation be interrupted to leave contracts in an inconsistent state?
- Can the liquidation process be manipulated to receive assets without proper payment or steal user collateral?
- Can rounding errors in financial calculations lead to fund loss or undercollateralization?
- Can the admin execute unauthorized operations that violate protocol constraints or steal user funds?
- Can the supervisor role be exploited to bypass admin controls or execute malicious operations?
- Can the upgrade mechanism be exploited to deploy malicious code or bypass security checks?
- Can the Pyth oracle integration be manipulated to return incorrect price information?
- Can race conditions be exploited to corrupt contract storage or manipulate user positions?
- Are there gas-related vulnerabilities that could lead to transaction failures during critical operations?

Project Targets

The engagement involved reviewing and testing the targets listed below.

EVAA Contracts (from June 2 to June 6)

Repository	https://github.com/evaafi/contracts_internal/
Version	`9faa122`
Type	FunC
Platform	TVM

EVAA Contracts (from June 9 onwards)

Repository	https://github.com/evaafi/contracts_internal/
Version	`60d59bf`
Type	FunC
Platform	TVM

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:

- **Files in scope.** The scope for the current audit, as provided by the Something Labs LTD team, consisted of the following files and folders. Additionally, to analyze how the contracts work, we needed to interact with files outside of the scope list; these files were given a lower priority.
    - `contracts/core/*`
    - `contracts/logic/master-if-active.fc`
    - `contracts/logic/master-utils.fc`
    - `contracts/logic/user-utils.fc`
    - `contracts/logic/utils.fc`
    - `contracts/logic/tx-utils.fc`
    - `contracts/logic/user-revert-call.fc`
    - `contracts/logic/user-upgrade.fc`
    - `contracts/messages/upgrade-header.fc`
    - `contracts/storage/*`
    - `contracts/master.fc`
    - `contracts/user.fc`
- **Supply and withdrawal flows.** We reviewed the contract input validations for the supply and supply-withdrawal operations, the interaction with external Pyth oracles when applicable, the intra- and inter-contract interactions between master and user contracts, and the balance updates after the operation executed.
- **Liquidation flows.** Similar to the previous flows, we reviewed input values boundaries, balance updates, correct calculations for the incentives, interest rates and fees. We assessed the interaction between master and user contracts, and between the master and Pyth oracles.
- **Privileged operations.** We reviewed the roles of the administrator account and the supervisors. We checked address validations before operations were executed, and normal accounts can execute privileged operations.
- **Arithmetic calculations.** We reviewed the interest rate calculations, liquidation fees and reserves calculations, and dynamic parameters updates. We reviewed rounding directions for off-by-one errors.
- **Upgrade operations.** For upgrade operations, we reviewed the sequence of messages required to perform the upgrade, the update of asset parameters, the upgrade and initialization of new versions of user contracts, and whether upgrade operations can trigger race conditions.

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:

- **External libraries.** All three files in the `external/` folder were considered to be libraries, and therefore were deprioritized for the scope of the audit. It was noted that the `stdlib.fc` file was modified from the canonical version.
- **Previous audit results.** We did not review issues mentioned in previous audit reports (Quantstamp) in this audit.
- **Third-party protocols.** We did not review the security of integrations with third-party protocols that may interact with EVAA contracts.

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	We identified multiple rounding issues: incorrect rounding in borrow amount calculations (TOB-EVAA-4) and inconsistent rounding for negative values (TOB-EVAA-9), both of which systematically favor borrowers over the protocol. The codebase lacks standardized rounding libraries and comprehensive tests for mathematical operations. The team should implement conservative rounding policies, add property-based testing for financial calculations, and establish formal verification for critical functions. The test suite should be expanded to include unit tests for arithmetic functions.	**Moderate**
Auditing	Critical actions, such as administrative interactions and asset movement operations, emit events for offchain monitoring. However, the team has not provided information about the existence of an offchain monitoring system and what operations it can perform. The team mentioned that they do not have a documented and tested incident response plan; we have provided a list of recommendations to implement one in appendix F.	**Moderate**
Authentication / Access Controls	Privileged functions have access control by address, and two different administrative roles are defined: administrator and supervisor. The supervisor role can be assigned and revoked by the administrator to any address. The team stated that the administrator address is a multisignature wallet; however, no available documentation supports that claim or clarifies the requirements for supervisor addresses. Appendix D contains a set of recommended best practices when working with multisignature wallets.	**Moderate**

Complexity Management	At a high level, the codebase is large, complex in nature, and difficult to maintain. In general, functions take more arguments than are required for them to operate. It is common to find several local variables that are not used. Code documentation can be improved, as there is no standard style for code comments. Most functions lack specific unit tests that ensure a correct implementation and parameter bounds checking.	**Weak**
Decentralization	Contracts are upgraded by the administrative account and do not allow users to opt out of the upgrade. User contracts are obligated to upgrade to interact with the master contract. There is no documentation about the administrative account privileges. The administrative account can unilaterally change system parameters after users open a position (TOB-EVAA-8), but in the current state of the contracts, privileged accounts cannot access user funds. From the user's point of view, usage of the contracts is decentralized in that it requires no permission or authorization.	**Weak**
Documentation	The EVAA website has user-facing documentation, but technical documentation is insufficient. Code comments are scarce, but some comments highlight important information or design decisions. This information should be incorporated into the technical documentation or description of the system. The team provided diagrams that show the complete flows for the protocol's functions, external oracle integration, and inter-contract messaging. These diagrams should also be part of the public documentation.	**Moderate**
Low-Level Manipulation	All low-level operations are delegated to libraries. The main protocol code is written in FunC.	**Satisfactory**
Testing and Verification	Running tests requires special instructions not found in the documentation. The provided test structure has 49 test suites with a total of 229 test cases for all most common use cases, and all	**Not Applicable**

	of the tests pass. However, both the test suite and the tests' coverage can be improved, as not all code paths are tested. There are no unit tests for all functions, and integration tests have coverage gaps. Appendix E contains recommendations for improving the test suite.	
Transaction Ordering	As TON is an asynchronous network, this category refers to risks associated with message ordering and race conditions. The protocol comprises several contracts, and it interacts with external parties such as Pyth, which introduces a risk of race conditions. Although several mitigations are in place, we still found the potential for a race condition in the system (TOB-EVAA-6).	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	The pause mechanism does not check for correct opcodes	Data Validation	**Medium**
2	New upgrade configuration is not written to storage	Data Validation	**Medium**
3	TON supply amount is locked when the gas check fails	Data Validation	**Medium**
4	Borrow amounts are incorrectly rounded	Data Validation	**Informational**
5	Origination fee application after collateralization check can cause undercollateralization	Data Validation	**Informational**
6	Supply cap bypass via return_repay_remainings_flag race condition	Timing	**Informational**
7	Insufficient fee validation leading to user contract state lock	Data Validation	**High**
8	Open positions can be affected by liquidation threshold changes	Data Validation	**Informational**
9	Incorrect rounding in present/principal value calculations for negative values	Data Validation	**Informational**

Detailed Findings

1. The pause mechanism does not check for correct opcodes

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EVAA-1
Target: `contracts_internal/contracts/logic/master-if-active-check.fc`	

Description

The protocol's pause mechanism relies on the `if_active_process` function to block user-initiated operations when the system is paused. The function checks if the operation code handles TON or Jettons, and returns the correct asset to the user in the case where the system is paused.

However, the current implementation of the opcode check looks for the deprecated `op::withdraw_master` opcode, and does not check for the new `op::supply_withdraw_master` opcode. This can result in users performing supply-withdraw operations when the system is inactive or paused.

```
if ((op == op::supply_master)
  | (op == op::withdraw_master)
  | (op == op::liquidate_master)
  | (op == op::idle_master))
  ;; ^ one of the "front-facing" op-codes, which initiate new operations
{
  ;; Stop processing and return TONs back to owner
  send_message(
    sender_address,
    0,

begin_cell().store_op_code(error::disabled).store_query_id(query_id).store_ref(begin
_cell().store_slice(in_msg_body).end_cell()).end_cell(),
    sendmode::CARRY_ALL_REMAINING_MESSAGE_VALUE
  );
  return ret::stop_execution;
} elseif (op == jetton_op::transfer_notification) {
```

Figure 1.1: The deprecated opcode check in contracts_internal/contracts/logic/master-if-active-check.fc#L23

Exploit Scenario

During a critical security incident, Alice, the protocol administrator, finds a potential issue in the lending pool logic and immediately pauses the protocol to avoid further damage. However, Bob discovers that he can still interact with the protocol by using the `op::supply_withdraw_master` operation. Bob continues to exploit the lending pools while legitimate users believe that the protocol is safely paused, potentially allowing him to drain protocol reserves before the team can implement a proper fix.

Recommendations

Short term, replace the `op::withdraw_master` check with an `op::supply_withdraw_master` check in the `if_active_process` function.

Long term, ensure that the test suite has enough coverage for all possible operations in the contracts, with positive and negative test cases.

2. New upgrade configuration is not written to storage

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EVAA-2
Target: `contracts_internal/contracts/core/master-admin.fc`	

Description

The protocol's upgrade process consists of three steps, represented by three opcodes:

- `init_upgrade_process`, which sets the new master and user code to storage and starts the upgrade timer
- `disable_contract_for_upgrade_process`, which ensures that no new operations are started and pending ones are completed before committing to the upgrade
- `submit_upgrade_process`, which sets the new master contract code, stores code versions, and performs the necessary checks to ensure that the upgrade succeeds

In the last step, a `new_upgrade_config` variable is generated to emit the upgrade log, but it is not saved to storage. This prevents the `master_version`, `user_version`, and `user_code` variables in storage from being updated, and the rest of the upgrade fields from being cleaned up correctly.

Since the current upgrade will be run using the v6 upgradeability feature, this can become an issue in the v7 to v8 upgrade. However, even in the case where the user contract code upgrade fails, the master contract code can still be upgraded, and a recovery function can be added to avoid losing control of the master contract.

```
() submit_upgrade_process  (
  [...]
) impure inline {
    throw_unless(error::message_not_from_admin, slice_data_equal?(sender_address,
admin));
    (
      int master_version, int user_version,
      int timeout, int update_time, int freeze_time,
      cell user_code,
      cell new_master_code, cell new_user_code
    ) = unpack_upgrade_config(upgrade_config);
    [...]
    cell new_upgrade_config = pack_upgrade_config(
```

```
      master_version, user_version,
      timeout, 0, 0,
      user_code,
      null(), null()
    );

    [...]

    on_upgrade(my_balance, msg_value, in_msg_full, in_msg_body);

    cell new_store = get_data();

    cell log_data = begin_cell()
      .store_uint(log::submit_upgrade, 8)
      .store_uint(now(), 32)
      .store_ref(begin_cell()
          .store_ref(upgrade_config)
          .store_ref(old_code)
          .store_ref(old_store)
        .end_cell())
      .store_ref(begin_cell()
          .store_ref(new_upgrade_config)
          .store_ref(new_master_code)
          .store_ref(new_store)
        .end_cell())
      .end_cell();
    emit_log_simple(log_data);

    recv_internal(my_balance, msg_value, in_msg_full,
begin_cell().store_op_code(op::do_data_checks).store_query_id(query_id).end_cell().b
egin_parse()
    );
    return ();
}
```

Figure 2.1: The `new_upgrade_config` variable in `contracts_internal/contracts/core/master-admin.fc#L432–L495`

Exploit Scenario

The EVAA team finds a bug in the current version of the user contract. As the contracts are upgradeable, they start the procedure for upgrading the master contract's storage with the new version of the user contract.

After finishing the upgrade, the EVAA team notices that the user contracts have not been upgraded because the master contract's storage was not persisted in the upgrade process. This effectively disallows upgrading the user contract.

Recommendations

Short term, persist the `new_upgrade_config` variable to the master contract's storage.

Long term, ensure that the test suite thoroughly checks the master contract's storage and that both master and user contracts' code changes on each upgrade process.

3. TON supply amount is locked when the gas check fails

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EVAA-3
Target: `contracts_internal/contracts/core/master-supply.fc`, `contracts_internal/contracts/core/master-supply-withdrawal.fc`	

Description

When a user supplies TON to the protocol, the supply amount can become permanently locked in the contract if the gas check fails. This occurs because the code reserves the supply amount before performing the gas check, preventing the refund message from accessing the reserved funds.

In the supply-withdrawal flow, when a user supplies TON, the code first calls `raw_reserve(supply_amount, reserve::AT_MOST + 2)` in the `supply_withdraw_ton` function to reserve the supply amount. If the subsequent gas check in the `master_core_logic_supply_withdraw` function fails (`msg_value < enough_fee`), the code attempts to refund the TON to the user by sending a message with `message_send_mode`. However, since the supply amount is already reserved, the refund message cannot access these funds, effectively locking them in the contract.

This issue could lead to users losing their TON if they do not provide enough gas for the transaction, as the reserved funds cannot be refunded. The same issue exists in the supply flow, where TON can be locked if the gas check fails.

```
cell master_core_logic_supply_withdraw(
    [...]
) impure {
    [...]
    int enough_fee = supply_withdraw_min_attachment(fwd_fee,
supply_withdraw_user_message) + ton_amount_for_repay_remainings;

    if (msg_value < enough_fee) {
        cell body =
begin_cell().store_op_code(error::supply_withdraw_master_transaction_fees).store_que
ry_id(query_id).store_ref(custom_response_payload).end_cell();
        if (supply_asset_id == constants::ton_asset_id) {
            send_message(
                initial_sender,
                0,
                body,
                message_send_mode ;; if ton => we send 128 + raw reserve
```

```
            );
        } else {
            [...]
        }
        return asset_dynamics_collection;
    }

    send_message_to_evaa_user_sc(
        BLANK_CODE(), user_version, user_code, initial_sender,
        supply_withdraw_user_message, message_send_mode, subaccount_id
    );

    return asset_dynamics_collection;
}


cell supply_withdraw_ton(
    [...]
) impure inline {
    raw_reserve(supply_amount, reserve::AT_MOST + 2);
    return master_core_logic_supply_withdraw(
        [...]
    );
}
```

Figure 3.1: The TON refund logic in supply-withdrawal flow in contracts_internal/contracts/core/master-supply-withdrawal.fc#L40–L97

Exploit Scenario

Alice wants to supply 100 TON to the protocol. She sends a transaction with 100 TON but includes only enough gas for a simple transfer, not accounting for the protocol's gas requirements. When the transaction executes, the `supply_withdraw_ton` function first reserves her 100 TON using `raw_reserve`. Then the `master_core_logic_supply_withdraw` function performs the gas check and fails due to insufficient gas. The code attempts to refund the 100 TON to Alice, but this fails because the funds are already reserved and cannot be accessed by the refund message. As a result, Alice's 100 TON becomes permanently locked in the contract with no way to recover it.

Recommendations

Short term, move the `raw_reserve` call after the gas check in the `master_core_logic_supply_withdraw` function.

Long term, implement a consistent pattern for handling gas checks and refunds across all asset types, and consider adding a recovery mechanism for locked funds in case similar issues occur in the future.

4. Borrow amounts are incorrectly rounded

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EVAA-4
Target: `contracts_internal/contracts/logic/user-utils.fc`	

Description

Borrow amounts represent user debt, and therefore must be rounded up to avoid understating user liabilities. In FunC, it is recommended to use ceiling rounding in the `muldivc` function for these calculations.

These small rounding errors can eventually add up and lead to a calculation mismatch between the internal accounting and the assets in custody.

Multiple borrow amount calculations throughout the `user-utils.fc` file fail to round up consistently, in particular:

- Line 164, in the `account_health_calc` function
- Line 231, in the `is_liquidatable` function
- Line 280, in the `get_available_to_borrow` function
- Line 354, in the `get_aggregated_balances` function

Recommendations

Short term, use `muldivc` instead of `muldiv` or normal multiplication and division when calculating user debt.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

5. Origination fee application after collateralization check can cause undercollateralization

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EVAA-5
Target: `contracts_internal/contracts/core/user-supply-withdrawal.fc`	

Description

The origination fee for new borrowings is applied after the collateralization check, which can result in undercollateralized positions immediately after borrowing.

In the `supply_withdraw_user_process` function, when a user withdraws an asset and the withdrawal creates a borrow position, the code first calls `is_borrow_collateralized` function to verify that the user has sufficient collateral.

```
(int borrow_is_collateralized, int enough_price_data) =
is_borrow_collateralized(asset_config_collection, asset_dynamics_collection,
user_principals, prices_packed);
if (borrow_is_collateralized) {
    [...]
    if (borrow_amount_principal > 0) {
        ;; borrow
        ;; so, we need to apply origination_fee to borrow amount
        int amount_to_borrow_in_present = - present_value(s_rate, b_rate, -
borrow_amount_principal);
        int origination_fee_taken =
amount_to_borrow_in_present.muldiv(origination_fee,
constants::origination_fee_scale);
        present -= origination_fee_taken;

        withdraw_new_principal = principal_value(s_rate, b_rate, present);

        (borrow_amount_principal, reclaim_amount_principal) =
around_zero_split(withdraw_new_principal, old_principal);

        user_principals~set_borrow_position(withdraw_asset_id,
withdraw_new_principal, origination_fee_taken, amount_to_borrow_in_present,
get_last_mc_block_seqno());
```

Figure 5.1: The collateralized check and origination fee logic in `contracts_internal/contracts/core/user-supply-withdrawal.fc#L124–L138`

However, the origination fee is deducted from the user's principal only after this check passes. This means that if a user borrows exactly up to their collateral limit, the subsequent

deduction of the origination fee will push their position below the required collateralization threshold, creating an unsafe state that violates the protocol's safety guarantees. Depending on the value of `origination_fee`, it could even push the user's position to the liquidatable threshold, making them immediately vulnerable to liquidation.

Recommendations

Short term, move the origination fee calculation and deduction before the `is_borrow_collateralized` check, or modify the collateralization check to account for the origination fee that will be applied.

Long term, expand unit tests to verify that no borrow transaction can result in an undercollateralized position after all fees are applied.

6. Supply cap bypass via return_repay_remainings_flag race condition

Severity: **Informational**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-EVAA-6
Target: `contracts_internal/contracts/core/user-supply.fc`	

Description

The `return_repay_remainings_flag` functionality creates a race condition that allows attackers to bypass the maximum token supply cap.

When `return_repay_remainings_flag = true`, the master contract does not update `awaited_supply` to account for pending supplies, assuming that this flag only allows repayment of existing borrowings without new supply. However, this assumption is flawed because the protection logic in `user-supply.fc` is ineffective for fresh accounts with no existing borrows.

The code checks for both `repay_amount_principal > 0` and `supply_amount_principal > 0`, but for fresh accounts where `repay_amount_principal = 0`, the logic falls into the `else` block where the max cap check is performed:

```
if((return_repay_remainings_flag == true) & (repay_amount_principal > 0) &
(supply_amount_principal > 0)) {
  total_supply += 0;
  total_borrow -= repay_amount_principal;
  new_principal = 0;
}
else {
  total_supply += supply_amount_principal;
  total_borrow -= repay_amount_principal;

  ;; max cap check and negative new total borrow check (only need this check in case
of return_repay_remainings_flag == false)
  if (((total_supply > max_token_amount) & (max_token_amount != 0) &
(supply_amount_principal > 0)) | (total_borrow < 0)) {
    reserve_and_send_rest(
      fee::min_tons_for_storage,
      master_address,
      pack_supply_fail_message(
        query_id, owner_address,
        asset_id, supply_amount_current,
        custom_response_payload, subaccount_id, return_repay_remainings_flag,
initial_sender
```

```
      )
    );
    return ();
  }
}
```

Figure 6.1: The logic for the return repay remaining flag in contracts_internal/contracts/core/user-supply.fc#L56–L78

However, because the `total_supply` is calculated in the master contract using `awaited_supply`, and `awaited_supply` is not updated when `return_repay_remainings_flag = true`, the max cap check uses stale data.

```
int total_supply_principal_with_awaited_supply =
  total_supply_principal + principal_value_supply_calc(s_rate, awaited_supply);

cell supply_user_message = pack_supply_user_message(
  query_id,
  asset_id, amount,
  s_rate, b_rate,
  dust, max_token_amount,
  total_supply_principal_with_awaited_supply, total_borrow_principal,
  tracking_supply_index, tracking_borrow_index,
  custom_response_payload,
  subaccount_id, return_repay_remainings_flag, custom_payload_recipient,
custom_payload_saturation_flag, initial_sender
);

int awaited_supply_with_incoming_amount = return_repay_remainings_flag ?
awaited_supply : awaited_supply + amount;
```

Figure 6.2: The `total_supply` calculation and `awaited_supply` update in contracts_internal/contracts/core/master-supply.fc#L36–L50

Due to TON's asynchronous nature, multiple concurrent supply transactions can be initiated before any complete, all seeing the same stale `awaited_supply` value when performing max cap checks. This allows attackers to inflate the token supply significantly beyond protocol limits, potentially destabilizing the economic model and risk management assumptions.

Recommendations

Short term, always update `awaited_supply` regardless of the `return_repay_remainings_flag` value.

Long term, implement explicit cap validation in the master contract before forwarding transactions to user contracts. This will ensure consistent enforcement regardless of user-level logic paths.

7. Insufficient fee validation leading to user contract state lock

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EVAA-7
Target: `contracts_internal/contracts/core/master-liquidate.fc`	

Description

The liquidation fee validation logic contains a flaw that allows attackers to permanently lock user funds by triggering incomplete liquidation processes. The current implementation validates only that `msg_value` is enough to cover `enough_fee` without accounting for the additional `TON_reserve_amount` required for successful transaction completion.

```
if ((min_collateral_amount > max_allowed_liquidation) | (msg_value < enough_fee)) {
  [...]

  ;; Refund asset
  immediate_asset_refund(
    [...]
  );

  ;; Note that we don't break execution:
  ;; We update asset_dynamics_collection with new s/b-rates regardless
} else {
  raw_reserve(TON_reserve_amount, reserve::AT_MOST + 2);

  send_message_to_evaa_user_sc(
    BLANK_CODE(), user_version, user_code, borrower_address, ;; <- the meaning is of
owner_address
    liquidate_user_message,
    sendmode::CARRY_ALL_BALANCE,
    subaccount_id
  );
}
```

Figure 7.1: Insufficient fee validation in contracts_internal/contracts/core/master-liquidate.fc#L54–L87

The actual gas requirement for successful completion is `enough_fee + TON_reserve_amount`. This creates an exploitable window where transactions can pass validation but fail during execution, leaving user contracts in a locked state.

When a liquidation request is processed, the user contract increments its state counter to track ongoing operations:

```
user::storage::save(
  code_version,
  master_address, owner_address,
  user_principals,
  state + 1, ;; Increase ongoing liquidation count
  user_rewards, backup_cell_1, backup_cell_2
);
```

Figure 7.2: State increment during liquidation in contracts_internal/contracts/core/user-liquidate.fc#L323–L329

This state variable acts as a critical lock mechanism that prevents withdrawals when it is incremented:

```
if (state != user_state::free) {
    reserve_and_send_rest(
        fee::min_tons_for_storage,
        master_address,
        pack_supply_withdraw_unsatisfied_message(
            query_id, owner_address,
            supply_asset_id, supply_amount,
            ton_amount_for_repay_remainings, custom_response_payload, subaccount_id
        )
    );
    return ();
}
```

Figure 7.3: Withdrawal prevention logic in contracts_internal/contracts/core/user-supply-withdrawal.fc#L21–L32

An attacker can exploit this by sending liquidation requests with `TON_reserve_amount < msg_value < enough_fee + TON_reserve_amount`. The transaction passes the initial fee check and increments the user's state, but the liquidation process fails to complete due to insufficient gas. This leaves the user contract permanently locked with `state > 0`, blocking all future withdrawal attempts indefinitely.

Exploit Scenario

Bob wants to attack Alice's account and lock her funds permanently. He identifies that Alice has a liquidatable position and crafts a malicious liquidation request.

Bob sends a transaction with `msg_value = 0.51 TON`, which covers the `TON_reserve_amount` (`0.5 TON`) but falls short of the total requirement of `enough_fee + TON_reserve_amount` (`0.52 TON`). The master contract's fee validation passes since `0.51 TON > enough_fee` (`0.02 TON`), and the liquidation process begins.

Alice's user contract receives the liquidation message and increments its `state` counter from 0 to 1 to track the ongoing operation. However, when the master contract attempts to

complete the liquidation process, it runs out of gas due to insufficient funds. The liquidation fails to complete, but Alice's user contract never receives the completion message to decrement the `state` back to 0. As a result, Alice's account remains permanently locked with `state = 1`, indefinitely preventing her from withdrawing any of her supplied funds.

Recommendations

Short term, update the fee validation logic to include the TON reserve amount requirement.

Long term, implement comprehensive gas estimation to prevent partial execution scenarios that could leave contracts in inconsistent states.

8. Open positions can be affected by liquidation threshold changes

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EVAA-8
Target: `contracts_internal/contracts/core/master-admin.fc`	

Description

The `op::update_config` privileged operation allows the administrator account to update all configuration parameters for the master contract, including the liquidation threshold value for opened positions.

It is likely that some EVAA users will not monitor their positions at all times, so a change on the liquidation threshold can affect existing positions, making them partially or completely liquidatable. Users will not be aware of the parameter change, and therefore will be unable to modify their positions to avoid being liquidated.

Recommendations

Short term, consider alternatives to give users time to adapt to parameter changes. For example, use a timelock mechanism to update critical protocol parameters, ensuring that there is enough time for existing users to learn about the change, and adjust their positions accordingly.

Long term, ensure that the documentation and website clearly mention these risks to users. Design a communication strategy that can reach as many users as possible to communicate incoming changes.

9. Incorrect rounding in present/principal value calculations for negative values

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EVAA-9
Target: `contracts_internal/contracts/logic/utils.fc`	

Description

During the supply and withdrawal flows, the protocol calculates the new principal value of the user position by calculating the actual present value of the position, adding or removing from it the new deposit or withdrawal, and calculating the principal value of that updated present value. This process, consisting of multiplications and divisions, is subject to rounding errors.

These rounding errors can lead to situations where, if the supplied or withdrawn value is zero or very small, the resulting present value is incorrectly calculated. In particular, in some cases, the resulting principal value may be less than the original value for supply operations, or more than the original value for withdrawal operations.

The present value and principal value calculation functions use inappropriate rounding for negative principal values (borrows), creating a systematic bias that favors borrowers at the expense of the protocol. The `muldivc` ceiling function rounds negative values toward zero rather than away from zero, which is incorrect for conservative debt calculations.

The present value calculation function routes to different rounding implementations based on the sign of the principal value:

```
(int) present_value_borrow_calc (int index, int principal_value) inline {
  return muldivc(principal_value, index, constants::factor_scale);
}
[...]
(int) present_value(int s_rate, int b_rate, int principal_value) inline {
  if (principal_value >= 0) {
    return present_value_supply_calc(s_rate, principal_value);
  } else {
    return present_value_borrow_calc(b_rate, principal_value);
  }
}
```

Figure 9.1: Borrow present value calculation using ceiling division in `contracts_internal/contracts/logic/utils.fc#L11`

The issue arises because `muldivc` performs ceiling division, which rounds toward positive infinity. For negative values, this means rounding toward zero, effectively reducing the absolute value of debt amounts. For example, if a calculation results in -8.5, `muldivc` rounds it to -8 instead of the more conservative -9. This systematic undercalculation of debt favors borrowers by reducing their owed amounts, while the protocol and suppliers bear the cost of these rounding errors.

Recommendations

Short term, ensure that present and principal value calculations use the correct rounding function that rounds negative values away from zero for conservative debt calculations.

Long term, implement comprehensive unit testing of all rounding operations to ensure that they align with the protocol's risk management strategy, where debt calculations should be conservative and favor the protocol over individual users.

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 Issues

The following issues are not associated with specific vulnerabilities. However, resolving them can enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Remove unused constants.** The following constants are defined, but they are not used in the EVAA codebase:
  - `plugins/pyth/op_codes.fc`
    - `op::pyth_parse_unique_price_feed_updates`
  - `plugins/pyth/errors.fc`
    - `error::jetton_execution_crashed`, `error::unauthorized`, `error::incorrect_price_feeds_num`, `error::price_not_actual`, `error::pyth_error_received`, `error::custom_operation_error`, `error::wrong_pyth_operation_code_code`, `error::unknown_operation`, `error::liquidate_like_failure`, `error::not_implemented_yet`
  - `constants/constants.fc`
    - `constants::is_this_current_rollout`
  - `constants/fees.fc`
    - `fee::supply_fail_revert_user`, `fee::supply_success_revert_user`, `fee::withdraw_fail`, `fee::liquidate_user_message`
  - `constants/errors.fc:`
    - `error::withdraw_master_transaction_fees`, `error::idle_target_not_allowed`, `error::withdraw_collateralized_fake_sender`, `error::user_code_broken`, `error::user_code_broken_on_upgrade`, `error::user_code_broken_on_transaction`, `error::prices_incorrect_signature`
  - `storage/user-storage.fc:`
    - `user_state::withdrawing`

- **Remove unused local variables and function parameters.** A list of all unused variables and function parameters would be excessively long, so we have not included it in this appendix. However, we recommend removing all unused variables and function parameters to improve the quality of the codebase.
- **Remove misplaced `include` statement**. In `contracts/core/master-liquidate.fc`, there is an `include` statement for `master.fc`. Even though the compiler would detect and solve the problem, this creates an `include` loop, since `master` already includes `master-liquidate`.
- **Fix typographical errors.**
    - `new_asssets_config`, in `core/master-admin.fc`
    - `error::liqudation_execution_crashed`, in `constants/errors.fc`
- **Improve the error for revert calls.** Currently, when a user code upgrade fails, or when an unknown opcode is received in the user contract's `recv_internal`, the contract calls the `revert_call` function, which throws with the code `error::user_code_version_mismatch`. This is correct for the first case, but wrong for the second.

D. Security Best Practices for Using Multisignature Wallets

Consensus requirements for sensitive actions, such as spending funds from a wallet, are meant to mitigate the risks of the following:

- Any one person overruling the judgment of others
- Failures caused by any one person's mistake
- Failures caused by the compromise of any one person's credentials

For example, in a 2-of-3 multisignature wallet, the authority to execute a "spend" transaction would require a consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, the following conditions are required:

1. The private keys must be stored or held separately, and access to each one must be limited to a unique individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the cosigner) should refer to a preestablished policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The cosigner should also verify that the half-signed transaction was generated willingly by the intended holder of the first signature's key.

Requirement #3 prevents the cosigner from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the cosigner can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to cosign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- An allowlist of specific addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the cosigner to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be cosigned. If the signatory were under an active threat of violence, he or she could use a duress code (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willingly, without alerting the attacker.

E. Testing Improvement Recommendations

This appendix aims to provide general recommendations on improving processes and enhancing the quality of the EVAA Finance test suite.

Identified Testing Deficiencies

During the audit, we encountered a number of issues that could have been prevented or minimized through better test practices and improved test coverage (e.g., TOB-EVAA-1, TOB-EVAA-2, TOB-EVAA-3, TOB-EVAA-7). We identified several deficiencies in the test suite that could make further testing and development more difficult and thereby reduce the likelihood that the test suite will find security issues:

- The tests are based on helper functions that are easy to use and make the test files readable, but do not allow the test cases to easily modify message fields such as message value or custom payloads.
- Existing tests do not cover the entire codebase. Mutation testing reveals coverage gaps. The following is a list of testing-related issues:
    - Tests for functions that update storage do not verify that it was correctly updated after execution.
    - Tests usually do not account for custom or malformed data, or not all value ranges are tested. This results in several branches not being executed.
    - When evaluating success or failure of a test case involving messages, the tests evaluate the existence of the message and its opcode, but do not validate the message contents.
    - Message values are always sufficient for paying gas and storage fees, so some checks always pass. Tests do not account for passing of time, preventing proper storage fee accrual.
    - User contract testing is not as thorough as the testing for the master contract. Since the master contract is the only allowed message sender to the user contract, the coverage is limited. For example, no malicious or malformed messages are sent from the master contract to the user contract, and not all opcodes are tested.
    - No concurrent tests are implemented. This is a limitation of the test suite execution, but can hide issues when checking, for example, the user contract state variable.

- All testing flows should include invariant checks before and after the tests run to ensure consistent states.
    - For example, the codebase should include an invariant that checks the contract balance before and after a supply operation and ensures that it does not decrease. This can help unveil potential fee miscalculation errors, and ensure that at all times, contract balances are not unexpected after a message is processed.
    - Another invariant should check if the `state` variable of the user contract is correctly reset after liquidations.
- Several administrative operations, such as the withdrawal of reserves, are not tested. Tests for contract upgrades are located in a different repository.
- Some parts of the code, including constants, are unused. See the Code Quality Issues appendix for more information.

To address these deficiencies and improve the EVAA Finance protocol's test coverage and processes, we recommend that the team define a clear testing strategy and create guidelines on how testing is performed in the codebase. Our general guidelines for improving test suite quality are as follows:

1. **Define a clear test directory structure.** A clear directory structure helps organize the work of multiple developers, makes it easier to identify which components and behaviors are being tested, and gives insight into the overall test coverage.
2. **Write a design specification of the system, its components, and its functions in plain language.** Defining a specification can allow the team to more easily detect bugs and inconsistencies in the system, reduce the likelihood that future code changes will introduce bugs, improve the maintainability of the system, and allow the team to create a robust and holistic testing strategy.
3. **Use the function specifications to guide the creation of unit tests.** Creating a specification of all preconditions, postconditions, failure cases, entrypoints, and execution paths for a function will make it easier to maintain high test coverage and identify edge cases.
4. **Use the interaction specifications to guide the creation of integration tests.** An interaction specification will make it easier to identify the interactions that need to be tested and the external failure cases that need to be validated or guarded against, and it will help identify issues related to access controls and external calls.
5. **Use fork testing for integration testing with third-party smart contracts and to ensure that the deployed system works as expected.** Fork testing can be used to

test interactions between the protocol contracts and third-party smart contracts by providing an environment that is as close to production as possible. Additionally, fork testing can be used to identify whether the deployed system is behaving as expected.

6. **Use mutation testing to identify gaps in the test coverage and more easily identify bugs in the code.** Mutation testing can help identify coverage gaps in unit tests and help discover security vulnerabilities.

Directory Structure

Creating a specific directory structure for the system's tests will make it easier to develop and maintain the test suite and find coverage gaps. This section contains brief guidelines on defining a directory structure.

- Create individual directories for each test type (e.g., `unit/`, `integration/`, `fork/`, `fuzz/`) and for the utility contracts. The individual directories can be further divided into directories based on components or behaviors being tested.
- Create a single base contract that inherits from the shared utility contracts and is inherited by individual test contracts. This will help reduce code duplication across the test suite.
- Create a clear naming convention for test files and test functions to make it easier to filter tests and understand the properties or contracts that are being tested.

Unit Testing

We provide the following general recommendations for unit testing based on our findings:

- **Define a specification for each function** and use it to guide the development of the unit tests.
- **Improve the unit tests' coverage** so that they test all functions and contracts in the codebase. Use coverage reports and mutation testing to guide the creation of additional unit tests.
- **Use positive unit tests** to test that functions and components behave as expected. Ideally, each unit test should test a single property, with additional unit tests testing for edge cases. The unit test should test that all expected side effects are correct.
- **Improve the use of negative unit tests** by not defining test cases that pass on any failure within a test body; instead, each negative unit test should test for a specific failure case.

Integration and Fork Testing

Integration tests build on unit tests by testing how individual components integrate with each other or with third-party contracts. It can often be useful to run integration testing on a fork of the network to make the testing environment as close to production as possible and to minimize the use of mock contracts whose implementation can differ from third-party contracts. We provide the following general recommendations on performing integration and fork testing:

- **Use the interaction specification to develop integration tests.** Ensure that the integration tests aid in verifying the interactions specification.
- **Identify valuable input data for the integration tests** that can maximize code coverage and test potential edge cases.
- **Use negative integration tests**, similar to negative unit tests, to test common failure cases.
- **Use fork testing to build on top of the integration testing suite.** Fork testing will aid in testing third-party contract integrations and in testing the proper configuration of the system once it is deployed.
- **Enrich the forked integration test suite with fuzzed values and call sequences.** Even though in the current state of TON tooling there are no fuzzers that we know of, we recommend leveraging fuzzing tools in future stages of development, as they become available. This will aid in increasing code coverage, validating system-level invariants, and identifying edge cases.

Mutation Testing

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We recommend using mutation tools that can help detect redundant code, insufficient test coverage, incorrectly defined tests or conditions, and bugs in the underlying source code being tested.

F. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Something Labs LTD. will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

G. Rounding Guidance

This appendix provides guidance on how arithmetic calculations must be rounded in order for protocols to remain solvent. This guidance should be followed for all arithmetic operations relating to functionality such as token deposits, withdrawals, transfers, and fee calculations.

Rounding guidance is necessary because certain arithmetic operations lose precision; this means that the final result of the operation is an approximation instead of the true result. The difference between an approximated result and the true result is called epsilon. It is critical for protocols to be aware of where epsilon will manifest in arithmetic calculations and correctly account for epsilon to prevent insolvency.

Trail of Bits maintains roundme, a human-assisted tool for determining rounding directions for individual operations in an arithmetic formula. After reading the sections below, engineers should determine which direction high-level formulas should round in, then use roundme to determine the rounding directions of the individual arithmetic operations that make up the formula.

Recommendations for Determining Rounding Directions

- **Identify where funds are sent from the protocol to users and ensure that the associated operations round down their results.** This will minimize the number of tokens that are sent out to users.
- **Identify where funds are sent from users to the protocol and ensure that the associated operations round up their results.** This will maximize the number of tokens that are kept by the pool.
- **Identify where funds are transferred from the protocol for fees and ensure that the associated operations round up their results.** This will maximize the number of tokens that stay inside the protocol.
- **Trace each variable backward through the project to verify that the rounding directions used to generate it are consistent.** This will help reduce the time it takes to debug more complex rounding issues.

Reasoning about Rounding Directions

To determine whether an operation should round up or down, one must reason about how the outcome of the operation impacts token transfers.

Consider the following equation used by a vault contract to calculate how many `share` tokens a user should be credited for a deposit of `assets`:

$$sharesToMint = (\frac{assets * supply}{totalAssets})$$

In order for this calculation to benefit the vault, `sharesToMint` should be rounded down. Variables that need to be rounded down are annotated with (↘):

$$sharesToMint\searrow = (\frac{assets\searrow * supply\searrow}{totalAssets\nearrow})\searrow$$

These variables must now be traced back through the contract to verify that they are rounded correctly. Sometimes a system is designed in such a way that not all variables can be rounded correctly; in cases like this, the protocol must be redesigned to facilitate correct rounding.

Integer Arithmetic Primitives

In integer arithmetic, division operations lose precision due to rounding. Therefore, division operations need two primitive operators to correctly approximate results: one where the quotient is rounded up and one where the quotient is rounded down.

Rounding down is the default behavior:

$$div_{down}(a, b) = \frac{a}{b}$$

The following is one approximation for rounding up an integer division operation:

$$div_{up}(a, b) = \begin{cases} \frac{a}{b}, & a \bmod b = 0 \\ \frac{a}{b} + 1, & a \bmod b \neq 0 \end{cases}$$

div_{down} and div_{up} can then be used as primitives to build the following:

- $mulDiv_{up}$
- $mulDiv_{down}$

H. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From July 16 to July 18, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the EVAA Finance team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Apart from the issues identified in this report, we also performed a best-effort analysis of mitigations for six issues found by the EVAA Finance team. This analysis is provided in the Client Findings Fix Review Results section.

In summary, of the nine issues described in this report, EVAA has resolved seven issues and has elected not to resolve the remaining two issues. The EVAA team has added context for each issue it chose not to address.

For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	The pause mechanism does not check for correct opcodes	Medium	Resolved
2	New upgrade configuration is not written to storage	Medium	Resolved
3	TON supply amount is locked when the gas check fails	Medium	Resolved
4	Borrow amounts are incorrectly rounded	Informational	Resolved
5	Origination fee application after collateralization check can cause undercollateralization	Informational	Unresolved
6	Supply cap bypass via return_repay_remainings_flag race condition	Informational	Resolved
7	Insufficient fee validation leading to user contract state lock	High	Resolved

8	Open positions can be affected by liquidation threshold changes	Informational	Unresolved
9	Incorrect rounding in present/principal value calculations for negative values	Informational	Resolved

Detailed Fix Review Results

TOB-EVAA-1: The pause mechanism does not check for correct opcodes
Resolved in commit fec61ec. The `op::withdraw_master` check has been replaced with an `op::supply_withdraw_master_without_prices` check in the `if_active_process` function to ensure proper opcode validation during pause operations.

TOB-EVAA-2: New upgrade configuration is not written to storage
Resolved in commit f6f7c16. The `on_upgrade` function has been refactored to separate the V7 upgrade logic into a dedicated `on_upgrade_after_v7` function. The new upgrade configuration is now properly written to storage within the `on_upgrade_after_v7` function, ensuring configuration persistence across upgrades.

TOB-EVAA-3: TON supply amount is locked when the gas check fails
Resolved in commit d45dc35. The supply amount reservation logic was restructured to occur after the gas check rather than before. The implementation now performs two separate reservations: the first reserves the original balance, and the second reserves the supply amount after successful gas validation, preventing fund lockup during gas check failures.

TOB-EVAA-4: Borrow amounts are incorrectly rounded
Resolved in commit 895562e. The borrow amount calculations have been updated to use proper rounding with the `muldivc` function, ensuring accurate financial calculations and preventing potential precision-related vulnerabilities.

TOB-EVAA-5: Origination fee application after collateralization check can cause undercollateralization
Unresolved. The client provided the following context for this finding's fix status:

> *This won't trigger under-collateralisation, because the origination fee is normally between 0 % and 0.3 %. The gap between the Collateral Factor (CF) and the Liquidation Threshold (LT) is typically about 5–7 %. That's how we tune EVAA and what the current config reflects. There's also an off-chain agreement that the admin will never set CF ≥ LT − origFee and will always keep a healthy buffer between these parameters.*
>
> *So from the protocol-logic perspective everything is safe—the user won't get liquidated right after borrowing up to 100 % of their collateral.*

TOB-EVAA-6: Supply cap bypass via return_repay_remainings_flag race condition
Resolved in commit f275a38. The `awaited_supply` variable is now updated consistently regardless of the `return_repay_remainings_flag` value, eliminating the race condition that could allow bypassing supply caps.

TOB-EVAA-7: Insufficient fee validation leading to user contract state lock
Resolved in commit c15b380. The `msg_value` validation logic has been corrected to reduce the checked amount by `liquidate_incoming_amount`, ensuring proper fee validation and preventing contract state locks.

TOB-EVAA-8: Open positions can be affected by liquidation threshold changes
Unresolved. The client provided the following context for this finding's fix status:

> *This design reflects the operational structure of our protocol, where the administrator is authorized to adjust these configurations. All such changes are governed by an off-chain agreement, under which the administrator commits to only increasing thresholds. In the event that a decrease is necessary, we will provide advance notification to all users.*

TOB-EVAA-9: Incorrect rounding in present/principal value calculations for negative values
Resolved in commit 2df64c6. The calculation logic has been refactored to handle the sign separately from the absolute value. The function now performs calculations exclusively with positive values and subsequently applies the original sign to the result, ensuring accurate rounding behavior for negative values.

Client Findings Fix Review Results

1. Incorrect opcode handling in Pyth fail case
Resolved in commit f287231. The commit submits a patch for an issue related to an incorrect opcode being used when handling Pyth oracle failure cases. The opcode should be related to Jetton flows because when the TON flow fails, the Pyth contract sends the error directly to users rather than to the EVAA master contract for error handling. The fix involved correcting the opcode to properly handle the Jetton flow scenario, ensuring that error messages are routed appropriately in oracle failure situations.

2. Missing validation for pools without TON coin configuration
Resolved in commit 34fd49a. The client identified that the code would break if TON was missing from the asset configuration. This scenario had not been previously encountered, as all prior pools included TON in their configurations. The fix added validation checks for operations to reject TON transactions when no TON entry exists in `asset_config_collection`.

3. Missing validation for redeem asset ID existence
Resolved in commit 39c1708. The client added validation to ensure that withdrawal asset IDs exist in the asset configuration before processing. While the action would revert

anyway and no one would be able to redeem nonexistent assets, this early validation check prevents the transaction from beginning execution.

4. Insufficient TON reserve for transfer notifications
Resolved in commit 98981f1. The client identified an issue where insufficient TON was reserved to cover transfer notifications when processing repay remainings with `supply_asset_id` is TON. The fix involved correcting the raw-reserve flags throughout the codebase and applying the same logic correction to all affected locations. Additional test cases have been implemented to prevent regression of this issue.

5. Insufficient validation in ton_amount_for_repay_remainings check
Resolved in commit 8543904. The previous implementation would replace `ton_amount_for_repay_remainings` with the constant `fee_jetton_tx` when the value was zero, but this still allowed callers to pass values that were too small, resulting in "insufficient gas" errors on outgoing Jetton transactions. The new logic rejects negative or undersized values outright, ensuring that adequate gas is available for transaction completion.

6. Pyth fee calculation function refactor
Resolved in commit b9f53a1. The fee calculation function for Pyth operations has been refactored to improve code clarity and maintainability.

7. Prices can not be duplicated for different asset IDs
Resolved in commit ef9ea25. This pull request adds support for tokens not having oracle prices to reference other asset prices, allowing jUSDT/jUSDC to use USDT/USDC prices. Additional safeguards have been implemented to return user assets if Pyth oracle fails.

I. Fix Review File Hashes

The following table shows the SHA-256 hashes of the in-scope contracts that were analyzed for the fix review:

File Path	SHA-256 hash
contracts/core/master-add-to-reserve.fc	e220887cb3...ef08431bf6
contracts/core/master-admin.fc	7dc4bf5e8e...2983f2b175
contracts/core/master-liquidate.fc	72b4b63810...8fac8dbeb6
contracts/core/master-other.fc	1cecd2f2cb...9980b0802a
contracts/core/master-revert-call.fc	cda2202196...86c93b3f65
contracts/core/master-supply-withdrawal.fc	b4dd19e1b6...bda4afb3c5
contracts/core/master-supply.fc	22b9d91fc3...15c8301a6a
contracts/core/user-admin.fc	226aa869fd...8d204811c5
contracts/core/user-liquidate.fc	9f4be12419...cf6ce1d316
contracts/core/user-other.fc	e0fb36feb0...195897c49f
contracts/core/user-supply-withdrawal.fc	10ad57a0f5...d568f8471b
contracts/core/user-supply.fc	1bfd2ea7e9...ecf06adecc
contracts/logic/master-if-active-check.fc	5d9ac66245...d363a47f92
contracts/logic/master-utils.fc	09b5b01caf...c10aaf60ed
contracts/logic/tx-utils.fc	536f03dbfd...3f9382cb51
contracts/logic/user-revert-call.fc	2546b5dc1b...b6568b35ce
contracts/logic/user-upgrade.fc	618868e63c...2ebc66cc00
contracts/logic/user-utils.fc	69961c284c...55df58d282
contracts/logic/utils.fc	93be9a2c66...ac5e943bee
contracts/messages/upgrade-header.fc	1e930014d1...9a313230cb

contracts/storage/master-storage.fc	6df3ec5082...60199165d2
contracts/storage/master-upgrade.fc	f163448bc5...d344007de8
contracts/storage/user-storage.fc	989772c4f2...573d83849b
contracts/storage/user-upgrade.fc	5cbd1d8c4b...14fe192921
contracts/master.fc	c249593a6c...964c1ef968
contracts/user.fc	e0ad77de87...fbe938fa73

The following table shows the SHA-256 hashes of the contracts that were part of the fix review but were not in scope for the audit:

File Path	SHA-256 hash
contracts/blank.fc	fc384fccf2...f8bd71019e
contracts/constants/constants.fc	4dd80ba5c0...5ef076f46c
contracts/constants/errors.fc	a5f8f8a16a...dde1770865
contracts/constants/fees.fc	dde9363eb2...fa9d00804c
contracts/constants/logs.fc	35d6afe2f9...105b39e9ae
contracts/constants/op-codes.fc	3a1558f37f...86796d93b2
contracts/data/asset-config-packer.fc	d43953a088...8e256b3a7c
contracts/data/asset-dynamics-packer.fc	23ddd9d569...229e34899e
contracts/data/basic-types.fc	7d64df0315...e83590f103
contracts/data/oracles-info.fc	0e330f0f11...23b82f9587
contracts/data/prices-packed.fc	06dc1bc0f9...07c94982dd
contracts/data/universal-dict.fc	11045f9fbc...1073684b5a
contracts/external/openlib.fc	166d1bff36...1608f39d8e
contracts/external/stdlib.fc	e601d34962...05744b7676
contracts/external/ton.fc	4dea11b2a3...66d9568abc

contracts/logic/addr-calc.fc	9159781aad...1c5faf9189
contracts/logic/master-get-methods.fc	e14f2a3959...410a08ead5
contracts/logic/user-get-methods.fc	9cc216660b...6fdc6c5740
contracts/messages/admin-message.fc	af1474c28c...6a554f92d9
contracts/messages/idle-message.fc	5e7c4cbf51...25c1c3f279
contracts/messages/liquidate-message.fc	a99deb7a1e...9d762b0021
contracts/messages/supply-message.fc	2b815ae920...79d294d8eb
contracts/messages/supply-withdraw-message.fc	157e53e573...cad9d7e291
contracts/plugins/pyth/constants.fc	ff3e34d2c3...75bade9ee6
contracts/plugins/pyth/errors.fc	738ebe38cf...14ceac1ea2
contracts/plugins/pyth/op_codes.fc	1e219eebfd...df14fd1467
contracts/plugins/pyth/parse_price_feeds.fc	b7393ffc44...38901198e3
contracts/plugins/pyth/pyth_request.fc	114f7b7ef8...d2e907ab03
contracts/plugins/pyth/stdlib.fc	73f8493d8e...ac27a6c0b1

J. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

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.