Affluent TON DeFi Protocol

Security Assessment

February 26, 2025

Prepared for:
Hyungyeon Lee
Affluent

Prepared by: **Tarun Bansal, Guillermo Larregay, and Elvis Skoždopolj**

Table of Contents

Table of Contents 1

Project Summary 3

Executive Summary 4

Project Goals 7

Project Targets 8

Project Coverage 9

Codebase Maturity Evaluation 11

Summary of Findings 14

Detailed Findings 16

1. Risk of unexpected borrow amount because of race condition 16
2. Lack of validation for configuration values 18
3. Risk of loss of fees and interest due to incorrect rounding direction in accrue_interest 19
4. Lack of two-step process for critical role transfers 21
5. The vault manager can deposit user assets into a non-whitelisted pool 22
6. A malicious farming contract can be used to mint infinite liquid tokens 24
7. Anyone can unfarm assets in a user's STON.fi liquid token wallet 26
8. Users can prevent their locked assets in the STON.fi liquid token wallet from being seized 29
9. Vault contract can become frozen due to incorrect balance reserve amount 32
10. Rounding down in the repay operations allows users to repay less than they owe 34
11. Protocol fees can be set higher than 100% and transferred to any address 37
12. Hard-coded forward payload for jetton transfers from the Vault contract 39
13. Liquidators can seize non-collateral assets from lending pools 40
14. Vault managers can collude with pool owners to get interest-free loans 41
15. Multiple supply and withdrawal requests from the same user to a Vault contract will fail 42
16. sync_locked_asset_cash can be executed multiple times 44
17. Missing or incorrect gas checks 47

A. Vulnerability Categories 48

B. Code Maturity Categories 50

C. Code Quality Recommendations 52

D. Rounding Recommendations 54

E. Fix Review Results **55**
Detailed Fix Review Results 57
F. Fix Review Status Categories **60**
About Trail of Bits **61**
Notices and Remarks **62**

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Project Timeline

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

Date	Event
January 2, 2025	Pre-project kickoff call
January 14, 2025	Status update meeting #1
January 21, 2025	Delivery of report draft
January 21, 2025	Report readout meeting
February 26, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Affluent engaged Trail of Bits to review the security of the Affluent DeFi protocol, built on TON. The protocol consists of several smart contracts, including the lending pool, oracle, vault, and liquid token smart contracts. The lending pool allows users to deploy and participate in the lending and borrowing process for a specific pair, or several pairs, of assets. The prices of assets can be fetched via the on-chain or off-chain (RedStone) oracles. The liquid token contracts define liquid versions of LP jettons deposited into DeDust or STON.fi, which could be used as collateral in the lending protocol. The vault contracts allow users to deposit an asset in the vault; the vault manager allocates the deposited assets to lending pools to generate good yield for the depositors.

A team of three consultants conducted the review from January 6 to January 17, 2025, for a total of six engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. Our testing efforts focused on analyzing the access control system, input data validation, and error handling flow, and on identifying any race conditions among user actions, corruption of contract state, arithmetic operation precision loss, and vulnerabilities to denial-of-service attacks. We checked the gas calculations and checks for correctness and evaluated the scope of privileged actors to see whether they could steal assets from the protocol users.

Observations and Impact

The codebase is structured well and broken down into small smart contracts that handle limited functionality to manage complexity. However, the organization of the common code and library contracts can be improved to reduce redundant functions and code blocks. The documentation and inline code comments help developers and reviewers navigate the code and follow user action message flows through different protocol smart contracts.

We discovered two high-severity issues involving an insufficient access control check (TOB-FACT-5) and an incorrect assumption about a third-party contract's behavior (TOB-FACT-6). We identified several race conditions that could lead to unexpected protocol behavior (TOB-FACT-1, TOB-FACT-15, and TOB-FACT-16) and allow users to delay liquidation (TOB-FACT-8). We also found low- and informational-severity issues related to input data validation, use of incorrect functions or hard-coded values, arithmetic operation precision loss, incorrect gas checks, and inefficient access control and incentive mechanism design.

The exclusive use of truncated arithmetic indicates a lack of a systemic approach toward rounding direction selection. The rounding directions should be carefully selected for each arithmetic operation to benefit the protocol.

Additionally, the liquid token design has several flaws. The seize and unfarm operations in the liquid token jetton wallets have been decoupled. The liquid token wallet contract allows anyone to unfarm another user's farmed jettons to facilitate the liquidation of user assets when their loan becomes undercollateralized. These actions are decoupled from the lending protocol liquidation flow and allow anyone to perform them even outside of a liquidation process. Any actions that are intended to be used strictly for liquidations should be coupled to the liquidation flow in order to prevent malicious users from griefing honest liquidators. The liquid token contracts check if the balance is 0 instead of a Boolean flag to determine if a user operation is in progress, which introduces race condition–related issues. Also, the liquid token jetton master contract deviates from the jetton standard. It does not update the `total_supply` storage variable when minting or burning liquid tokens. Similarly, the contract depends on the transfer notification message for balance updates and does not safeguard against malicious third-party contracts, which can introduce security vulnerabilities.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Affluent take the following steps before deploying the protocol smart contracts in production:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.

- **Expand the documentation.** Expand the documentation to include information about the message TL-B scheme and validation logic at every stage. Create an access control system specification with roles and their privileges.

- **Analyze and document the effect of race conditions.** Document in diagrams all the different paths of user action flow, including the number of transactions and their state changes. Analyze these flow diagrams to detect race conditions and determine the scope of their effect, and document them to discover and resolve security issues.

- **Make the liquid token design robust.** Consider integrating the seize and unfarm operations to reduce the impact of race conditions. Determine if the liquid token wallet's liquidation operations can be more strongly coupled to the lending protocol liquidation flow. Safeguard the liquid token contracts against malicious third-party contracts.

- **Analyze the precision loss in arithmetic operations.** Determine the correct precision scale, and implement a systematic rounding direction strategy for all arithmetic operations to benefit the protocol.

- **Improve the test suite.** Consider invalid or malicious user inputs to test protocol stability against them. Reduce mock contracts and use multiple jettons to align the test suite with the production environment.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	5
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Authentication	1
Data Validation	12
Denial of Service	1
Timing	2
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Affluent lending protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a user create permanent bad debt?
- Can a user borrow more than their collateral?
- Can a user prevent liquidation?
- Can an attacker steal user deposits?
- Can an attacker prevent depositors from withdrawing their assets?
- Can a user create a loan for someone else?
- Can a user repay less than their borrowed amount?
- Can race conditions be exploited to take malicious actions such as the following?
    - Prevent liquidation
    - Prevent the seizing of underlying assets of the liquid tokens
    - Increase a user's balance or otherwise corrupt the contract storage
- Is it possible to trigger the `take_oracle_price` message with malicious parameters in order to manipulate other users' deposits and loans?
- Can the rounding directions of protocol operations be abused to gain assets, reduce interest, or reduce repayment amounts?
- Can attackers transfer their own liquid tokens to themselves to corrupt the liquid token balances or cause balance inflation?
- Can pools be reinitialized?
- Are the access controls effective?
- Can the on-chain oracle price be manipulated?
- Can the vault manager and/or owner steal assets from the vault?
- Can the protocol fee configurer steal assets or otherwise manipulate the lending pool?

Project Targets

The engagement involved reviewing and testing the following target.

factorial-core

Repository	https://github.com/factorial-finance/factorial-core
Version	Commit `9b67bc57`
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:

- **General:** We reviewed the whole codebase for common FunC and TON blockchain flaws, such as missing or incorrect gas checks, issues with bounced message handling, and message parsing and building errors. We checked the use of modifying functions, message and storage cell limits, error handling logic, maintenance of the contracts' TON balance, and contract deployment and initialization, all for correctness.

- **Lending pool:** We reviewed the message flow of all user actions, including `supply`, `withdraw`, `borrow`, `repay`, and `liquidate`. We checked whether user actions are completed and errors handled correctly without leaving the system in an inconsistent state. We checked whether users can lose their assets and attackers can steal user funds. We also looked for vulnerabilities of denial-of-service attacks on the protocol pool smart contracts.

- **Oracle:** The protocol uses RedStone price feeds and on-chain oracle contracts to fetch prices from on-chain exchanges. These prices are used to calculate users' risk ratios and leverage ratios. We checked the RedStone price feed data parsing logic and jetton decimal handling logic for correctness. We evaluated the oracle contracts for any risk of on-chain oracle price manipulations and their impact on the protocol. We also checked whether the oracle contracts can be used to bypass access control checks.

- **Vault:** The vault contracts accept user deposits, which a vault manager manages to generate yield from the lending pools. The vault owner manages a whitelist of lending pools in which the manager can deposit user assets. The vault manager can allocate funds among the whitelisted pools by setting their target weights or directly withdrawing and depositing them into the pools. We reviewed the vault contracts to check if the vault manager can steal user funds or divert them to a pool to financially benefit from it. We also checked whether users can withdraw more than their share of the vault or inflate their vault shares without depositing any assets.

- **Liquid tokens:** The protocol includes two liquid token contracts to facilitate the farming of DeDust and STON.fi LP tokens while they are used as collateral in the lending pools. We reviewed the liquid token contracts to check whether users can manipulate their liquid token balance by combining multiple user actions. We looked for race conditions affecting the syncing, farming, and seizing operations.

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:

- We reviewed the liquid token contracts for race conditions and improper state updates and validation. However, we could not fully triage all of the concerns we found during this review. The issues we discovered in the STON.fi liquid token jetton wallet indicate that more issues might be present in these contracts.

- We discovered multiple lower-severity issues arising from the use of truncating arithmetic. However, there is a chance that multiple actions that perform truncating arithmetic can be combined to extract profit from the protocol. We recommend that the Affluent team analyze the arithmetic precision and select explicit rounding directions for the arithmetic operations.

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 protocol uses simple arithmetic operations to compute user shares, token amounts, asset prices, risk ratios, and leverage ratios. The mathematical formulas are well documented. We identified rounding errors and precision loss in some of the calculations performed by the protocol, as described in TOB-FACT-3 and TOB-FACT-10. Rounding directions are not explicit; operations always round in the default direction. Performing a precision loss analysis to determine an effective precision scale for storing amounts, and applying the recommendations provided in appendix D, would help mitigate the system's loss-of-precision risks and help evaluate them more effectively.	**Weak**
Auditing	No external messages are emitted by the contracts, making it difficult to monitor events off-chain. We have no information about the existence of a monitoring system or an incident response plan.	**Missing**
Authentication / Access Controls	The protocol implements an effective access control system to prevent unauthorized access. All privileged functions have some form of access control following the principle of least privilege. The test suite includes test cases for positive and negative access control checks. However, we found an access control issue resulting from the use of an incorrect imported function (TOB-FACT-5). The system can benefit from additional documentation specifying the roles, their privileges, and the process of granting and revoking them.	**Moderate**
Complexity	In general, the codebase is well structured and broken	**Moderate**

Management	down into small contracts and functions. However, a lack of a well-defined convention for function and variable naming makes it hard to navigate and reason about the code. The common code for messages, custom types, and constants can be organized better to simplify import paths, reduce duplicate code, and make it easy to navigate the codebase.	
Decentralization	In the current state, privileged actors have complete control over the protocol parameters, configuration, and upgradeable contracts. Issues TOB-FACT-11 and TOB-FACT-14 show that privileged actors can use the protocol to their benefit and profit from it. However, the Affluent team stated that, initially, these privileged roles will be assigned to multisignature wallets with a timelock functionality. Eventually, the contracts will be immutable, and the ownership of contracts will be dropped, making the protocol decentralized.	**Weak**
Documentation	The code lacks inline documentation of files and functions. For example, structures are manually packed and unpacked into cells, but this is not explained in code comments; such comments would significantly clarify the code. The provided documentation is good for a high-level understanding of how the protocol works but can be improved with the addition of more technical details. The mermaid diagrams help explain the different actions, but they can be improved to show what data is exchanged in each message. There is no information about TL-B schemes for the different messages or data types. The user-facing documentation can also benefit from inclusion of information about system limitations and potential risks associated with the asynchronous nature of user actions.	**Moderate**
Low-Level Manipulation	There are no instances of Fift assembly code in the protocol files.	**Not Applicable**
Testing and Verification	The test cases do not cover the whole protocol codebase or features, and the test coverage is not 100% for several	**Weak**

	test files. Considering issue TOB-FACT-2, we recommend improving the test suite by adding adversarial cases such as invalid configurations and incorrect parameters for actions. Additionally, the tests should use values that closely match the ones that will be used in production. For instance, the interest accrual tests use a time delta of one year, which is unrealistic. Also, issue TOB-FACT-16 could have been easily discovered by using various values in the tests, particularly zero.	
Transaction Ordering	Even though the TON blockchain network does not allow users to time when transactions are executed, race conditions among user actions can affect the protocol's stability. The Affluent team minimized smart contract state synchronization risks by having the protocol collect all the required data and enforcing all system checks in one place. However, the risks of race conditions are not clearly identified or documented. We identified race conditions that could enable timing attacks as described in issues TOB-FACT-1 and TOB-FACT-8. Creating a detailed user action flow diagram with the number of transactions and state change from every transaction can help discover race conditions.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Risk of unexpected borrow amount because of race condition	Timing	Low
2	Lack of validation for configuration values	Data Validation	Informational
3	Risk of loss of fees and interest due to incorrect rounding direction in accrue_interest	Data Validation	Low
4	Lack of two-step process for critical role transfers	Data Validation	Informational
5	The vault manager can deposit user assets into a non-whitelisted pool	Data Validation	High
6	A malicious farming contract can be used to mint infinite liquid tokens	Data Validation	High
7	Anyone can unfarm assets in a user's STON.fi liquid token wallet	Authentication	Medium
8	Users can prevent their locked assets in the STON.fi liquid token wallet from being seized	Timing	Medium
9	Vault contract can become frozen due to incorrect balance reserve amount	Denial of Service	Low
10	Rounding down in the repay operations allows users to repay less than they owe	Data Validation	Low
11	Protocol fees can be set higher than 100% and transferred to any address	Data Validation	Low
12	Hard-coded forward payload for jetton transfers from the Vault contract	Undefined Behavior	Informational

13	Liquidators can seize non-collateral assets from lending pools	Data Validation	**Informational**
14	Vault managers can collude with pool owners to get interest-free loans	Data Validation	**Informational**
15	Multiple supply and withdrawal requests from the same user to a Vault contract will fail	Data Validation	**Informational**
16	sync_locked_asset_cash can be executed multiple times	Data Validation	**Informational**
17	Missing or incorrect gas checks	Data Validation	**Informational**

Detailed Findings

1. Risk of unexpected borrow amount because of race condition	
Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-FACT-1
Target: `contracts/core/pool.fc`, `contracts/core/pool/handler.fc`	

Description

There is a race condition between the borrow action messages and interest accrual action that can result in unexpectedly higher borrow amounts when users specify a borrow share value. The borrower would have to pay a higher interest amount on this higher loan, so they would suffer unexpected financial loss.

The `Pool` contract allows users to specify the borrow share value when borrowing assets by setting the `is_share` field of the `transfer_out_params` of the borrow message to `true`.

```
if ((op == op::withdraw) | (op == op::borrow)) {
    cell oracle_params = in_msg_body~load_maybe_ref();
    try {
        ;; Parse the message body
        (slice asset_address, int is_share, int value, _, _, int forward_ton_amount,
_) = in_msg_body.preload_transfer_out_params();
        (var asset, int asset_found?) = pool::assets.get_asset?(asset_address);
    ...
```

Figure 1.1: A snippet of the borrow operation handler of the `Pool` contract (`contracts/core/pool.fc#L166–L171`)

The borrow message handler validates the user inputs and sends a message to the oracle contract to fetch the asset prices. Some asset prices are available from the RedStone price feed, and some are fetched from the on-chain Price Aggregator contract. The oracle contract then sends the prices back to the `Pool` contract, and the `Pool` contract forwards the prices with the borrow message to the `Account` contract. The `Account` contract validates the user's risk and leverage ratios and sends the borrow success message to the `Pool` contract. The `Pool` contract then calculates the asset amount to be transferred to the user for the specified share value based on the total borrow amount and total borrow shares of the pool.

However, the last message of this flow (the success message) may reach the `Pool` contract after another borrow action that started before it because of the difference in the oracle contract message path. The `accrue_interest` operation can also be executed between the borrow action start and end message to the `Pool` contact.

In this case, the interest accrual process would update the pool's total borrow amount without updating the pool's total borrow shares, resulting in the computation of a higher asset amount for the same number of borrow shares. Therefore, a user may get a higher asset amount than expected because of the race condition between the borrow action messages and interest accrual action.

Exploit Scenario
A USDT pool has a total borrow amount of 100,000 USDT and total borrow shares of 100,000. Alice wants to borrow 10,000 USDT from the pool and sends a borrow message with `is_share` set to `true` and `value` set to 10,000. When the `Pool` contract processes Alice's message, the borrow amount is computed to be 10,000 USDT, as expected by Alice. Bob executes an `accrue_interest` message before Alice's borrow action completes. The interest accrual results in a total borrow amount for the pool of 100,100 USDT and a total of 100,000 borrow shares. Now, when Alice's borrow action message is sent from the `Account` contract to the `Pool` contract, the USDT amount for 10,000 borrow shares is calculated as 10,010 USDT. Alice gets 10,010 instead of the expected 10,000 USDT and has to pay interest on an extra 10 USDT.

Recommendations
Short term, make one of the following changes:

- Add minimum and maximum borrow amount parameters to the borrow operation message to allow users to set slippage limits.
- Remove the functionality allowing users to specify a borrow shares amount.

Long term, analyze race conditions among different user actions and message flows to discover and prevent unexpected protocol behavior.

2. Lack of validation for configuration values	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACT-2
Target: Several targets	

Description
Several initialization and configuration values are missing validation checks.

- **Pool initialization:** When executing the `op::initialize` operation, the code checks that the given pool was not previously initialized. The Factory contract initializes all pools on deployment, but pools created by third parties may not have been initialized when deployed. Such a pool would pass the `op::initialize` check, so anyone can change the pool's initialization data, including the pool owner, and initialize it with this new data. To protect users, the Affluent team should only show pools deployed by Affluent in the UI. Alternatively, users should be aware that third-party pools that were not correctly deployed can pose a risk.

- **Pool liquidation close ratios:** The Factory contract and the pool initialization message handler are missing checks to ensure that the minimum liquidation close ratio is always less than the maximum.

- **Interest rate model (IRM) configuration:** The jump model interest rate calculation does not check the value of `kink_low` to ensure it is less than the value of `kink_high` for all assets.

- **Configuration values in general:** Throughout the codebase, certain values that can be set only once or changed only by privileged accounts are missing checks of their upper and lower bounds, allowing them to be set to incorrect values that could result in fund losses, incorrect calculations, or other situations affecting users. Some examples include values related to pool configuration and asset configuration.

Recommendations
Short term, add the validation checks described above to ensure that all configuration values and contract parameters have upper and lower bounds and that ranges are correctly ordered.

Long term, improve the test suite by adding cases to detect incorrect configurations.

3. Risk of loss of fees and interest due to incorrect rounding direction in accrue_interest	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-3
Target: `contracts/core/pool/asset.fc`	

Description

Due to the use of truncating arithmetic to calculate the accrued interest in the `accrue_interest` function, the calculation will result in zero if relevant values such as the total borrowed amount are sufficiently small.

The `accrue_interest` function calculates the accrued interest by multiplying the `time_delta` (in seconds) with the stored interest rate per second denominated in 10^{36} precision ($10^{36} = 100\%$) and the `total_borrow_amount` denominated in the decimal precision of the jetton (10^{9}). This intermediary value is divided by 10^{36}, as shown in figure 3.1:

```
int interest = muldiv(time_delta * asset.get_stored_interest_rate(),
asset.get_total_borrow_amount(), DECIMALS_OF_INTEREST);
int protocol_fee = muldiv(interest, protocol_fee_rate, DECIMALS_OF_PERCENT_TYPE);
```

Figure 3.1: The `accrue_interest` function in *`contracts/core/pool/asset.fc#L242–L243`*

However, with a sufficiently small interest rate, `time_delta`, and `total_borrow_amount`, the multiplication will result in a value less than 10^{36}, causing the result of the division to round down to zero.

Additionally, the protocol fee is calculated as a percentage of the result of the interest calculation. This means that the precision loss of the calculation of the protocol fee amount is even higher, causing the protocol to lose fees over time even when the precision loss of the accrued interest is not significant.

Exploit Scenario

Alice deposits some amount of collateral and borrows $1 * 10^{8}$ wrapped bitcoin (approximately $9,400 at the time of writing). The current utilization results in an interest rate of 5% per annum (approximately $1.585 * 10^{27}$ per second scaled to 10^{36} precision).

She calls the `accrue_interest` function in the next block, and it calculates the interest rate (assuming 5 seconds as block time):

$$\frac{\sim 1.585 * 10^{27} * 5 * 1 * 10^{8}}{10^{36}} \approx 0.792 \Rightarrow 0$$

This calculation truncates the result, resulting in zero interest accrued. If a user action executes `accrue_interest` in every block, zero interest will be accrued for a longer period of time, making loans interest-free for the borrowers, causing lenders to lose earned interest and the protocol to lose fees.

Recommendations

Short term, consider storing the interest and asset amounts with higher precision by using a scaling factor in order to minimize the cumulative loss of precision.

Long term, analyze the arithmetic formulas used in the protocol and consider implementing explicit rounding directions that favor the pool over the users. Update the testing suite by adding tests that use various `time_delta` values for interest accrual.

4. Lack of two-step process for critical role transfers	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACT-4
Target: `contracts/core/pool/handler_owner.fc`, `contracts/periphery/factory/factory/handler_owner.fc`	

Description

When called, the `set_owner` operation handler function immediately sets the owner of the Pool contract. The use of a single step to make such a critical change is error-prone; if the function is called with erroneous input, the results will be irrevocable.

```
if (op == op::set_owner) {
    slice new_owner = in_msg_body~load_msg_addr();
    pool::owner = new_owner;
    save_data();
    return true;
}
```

Figure 4.1: The `set_owner` operation handler of the pool contract (`contracts/core/pool/handler_owner.fc#L94–L99`)

The `set_fee_configurer` operation handler of the `Factory` and `Pool` contracts are also affected by the same issue.

The Affluent team explained to us that a multisignature contract with the timelock function is the owner of the `Pool` contract. This allows executors a chance to catch and correct any erroneous input for these critical functions. The team also specified that the ownership role is used during the testing phase and will be disabled in the protocol's production deployment.

Recommendations

Short term, use two steps for transferring critical roles: the first step to propose a new account for the role and the second step to accept the role from the new account itself.

Long term, identify and document all possible actions that can be taken by privileged accounts, along with their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

5. The vault manager can deposit user assets into a non-whitelisted pool

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACT-5
Target: `contracts/vault/share_vault/imports/handler_manager.fc`	

Description

The vault owner can remove a pool from the whitelist, but the vault manager can still deposit vault cash into the non-whitelisted pool. This allows the vault manager to divert all user assets to the non-whitelisted pool.

The `supply_to_pool` action handler in the `handler_manager` contract checks if the specified pool exists in the `vault::whitelisted_pools` dictionary, but it does not check if the `whitelisted?` field of the pool info is set to `true` before depositing into the pool. This allows the vault manager to deposit into non-whitelisted pools.

```
if (op == op::supply_to_pool) {
    slice pool_address = in_msg_body~load_msg_addr();
    throw_unless(error::not_whitelisted_pool,
vault::whitelisted_pools.is_whitelisted?(pool_address));
    throw_unless(error::not_enough_gas, msg_value > gas::min_supply_gas);

    int supply_amount = in_msg_body~load_coins();

    throw_unless(error::not_enough_cash, vault::cash >= supply_amount);
    throw_unless(error::not_enough_balance, vault::balance >= supply_amount);

    vault::balance -= supply_amount;
    send_factorial_supply_msg(query_id, vault::asset_wallet_address, pool_address,
vault::asset_address, supply_amount, vault::manager_address, manage_action_addr(),
0, msg::gas_refund_payload(vault::manager_address));
    send_excesses(query_id, sender_address, max(ctx::ton_balance_before_msg,
NANOTON_1e8));
    save_data();
    return true;
}
```

Figure 5.1: The `supply_to_pool` action handler of the `Vault` contract
`contracts/vault/share_vault/imports/handler_manager.fc#L21–L36`

Exploit Scenario

Bob, the vault owner, adds three pools to the `vault::whitelisted_pools` dictionary of the vault `VaultA`. After some time, Bob sets `PoolX` as a non-whitelisted pool by setting the `whitelisted?` field of the pool info slice stored as a value for the `PoolX` address in the dictionary to `false`. Eve, the vault manager, withdraws user assets from all other pools and deposits them into `PoolX`. This results in an unexpected financial loss for the vault depositors.

Recommendations
Short term, add a check to ensure that the entry found for the `pool_address` key in the `vault::whitelisted_pools` dictionary has the `whitelisted?` field set to `true`.

Long term, improve the test suite to check the effects of all privileged user actions on other privileged user actions.

6. A malicious farming contract can be used to mint infinite liquid tokens	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FACT-6
Target: `contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func`	

Description

A malicious farming contract can be used to send a `transfer_notification` message that mints new liquid tokens when the `unfarm` action is called to transfer the underlying assets deposited into the contract. Such a malicious contract would allow users to mint infinite liquid tokens.

The STON.fi liquid token jetton wallet contract allows users to deposit an underlying asset into a farming contract, using the `farm` action. This action reduces the `storage::locked_asset_cash` value but does not reduce the `storage::balance` and `storage::locked_asset_balance` values:

```
if (ctx.op() == op::farm) {
  throw_unless(error::unauthorized, equal_slice_bits(storage::owner_address,
ctx.sender_address()));

  ...

  int amount = in_msg_body~load_coins();
  slice response_address = in_msg_body~load_msg_addr();
  throw_unless(error::not_enough_balance, storage::balance >=
storage::locked_asset_balance);
  throw_unless(error::not_enough_locked_asset_cash, storage::locked_asset_cash >=
amount);
  send::stonfi::stake(amount, response_address, storage::balance);
  storage::locked_asset_cash -= amount;
  save_data();
  return ();
}
```

Figure 6.1: The `farm` action handler of the STON.fi liquid token wallet contract
`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L117–L134`

Users then use the `unfarm` action to withdraw the underlying assets from the farming contract. However, if a farming contract includes a nonzero `forward_ton_amount` value in the jetton transfer message, then this `unfarm` action will be processed by the liquid token wallet contract like any other deposit by the user and will mint new liquid tokens to the user. A malicious actor can create or upgrade a farming contract to send such a `transfer_notification` message to mint infinite liquid tokens by repeatedly executing the `farm` and `unfarm` actions.

Exploit Scenario
Eve creates a farming contract that rewards depositors with a good amount of assets to attract users and integrators. The Affluent team creates a liquid token contract for this farming contract. Eve upgrades the farming contract to send a `transfer_notification` whenever she executes the `unfarm` action on the liquid token wallet contract. Eve mints a large number of liquid tokens and deposits them as collateral into a Affluent lending pool. Eve then borrows a large amount of other assets from the lending pool and creates a permanent bad loan in the protocol.

Recommendations
Short term, implement a check to ensure that the underlying asset transfer from the farming contract does not mint new liquid tokens.

Long term, consider the malicious behavior that third-party contracts may take and minimize assumptions about their behavior to prevent unexpected protocol behavior.

7. Anyone can unfarm assets in a user's STON.fi liquid token wallet	
Severity: **Medium**	Difficulty: **Low**
Type: Authentication	Finding ID: TOB-FACT-7
Target: `contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func`	

Description

Anyone can execute the `unfarm` action when a user's liquid token balance is less than the `locked_asset_balance` value. This allows malicious actors to unfarm any user as soon as they deposit their liquid token to a Affluent lending pool.

The STON.fi liquid token Jetton wallet contract allows anyone to execute the `unfarm` action on a user's liquid token Jetton wallet contract when the user's liquid token balance becomes less than the value of the `storage::locked_asset_balance` storage variable:

```
if (ctx.op() == op::unfarm) {
  if (storage::connected_with_pool_account?) {
    var forward_payload =
begin_cell().store_op(op::unfarm).store_slice(ctx.sender_address()).store_slice(in_m
sg_body);
    send::account::provide_account_status(forward_payload);
    return ();
  }
  ifnot (equal_slice_bits(storage::owner_address, ctx.sender_address())) {
    throw_unless(error::enough_locked_asset_balance, storage::balance <
storage::locked_asset_balance);
  }

  slice nft_address = in_msg_body~load_msg_addr();
  send::stonfi::unstake(nft_address);
  return ();
}
```

Figure 7.1: *`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L136–L149`*

Additionally, when the `unfarm` action is called on a user who has their pool account connected to their liquid token wallet, the action implements a wrong balance check. After getting the pool account status, the action checks the value of `storage::balance` instead of `total_balance`.

```
if (ctx.op() == op::take_account_status) {
  throw_unless(error::unauthorized, equal_slice_bits(storage::pool_account_address,
ctx.sender_address()));
  cell account_status = in_msg_body~load_ref();
  storage::connected_with_pool_account? = true;
  if (slice_empty?(in_msg_body)) {
    save_data();
    return ();
  }

  (slice account_cs, int account_status_found?) =
account_status.adict_get?(storage::jetton_master_address);
  int total_balance = account_status_found? ? storage::balance +
account_cs~load_coins() : storage::balance;

  ...

  if (payload::op == op::unfarm) {
    slice sender_address = in_msg_body~load_msg_addr();
    ifnot (equal_slice_bits(storage::owner_address, sender_address)) {
      throw_unless(error::enough_locked_asset_balance, storage::balance <
storage::locked_asset_balance);
    }

    slice nft_address = in_msg_body~load_msg_addr();
    send::stonfi::unstake(nft_address);
    return ();
  }
  ...
```

Figure 7.2:
contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L230–L265

This action is intended to allow liquidators to unfarm and seize a user's assets when the user is liquidated in a lending pool. However, malicious actors can misuse this allowance to `unfarm` any user as soon as they deposit their liquid tokens into a lending pool or transfer them to another account, causing the user to lose farming rewards and defeating the purpose of using the liquid token contract.

Exploit Scenario

Alice provides liquidity to a STON.fi TON/USDT pool and gets 1,000 STON.fi liquidity tokens. She deposits the STON.fi liquidity tokens to her Affluent liquid token wallet and mints 1,000 liquid tokens. She executes the `farm` action on her liquid token wallet to transfer the STON.fi liquidity tokens to a farming contract to earn farming rewards. Then, she deposits her liquid tokens to a Affluent lending pool. As soon as she deposits her liquid tokens to a lending pool, Eve executes the `unfarm` action on Alice's liquid token wallet. After some time, Alice executes the `claim_rewards` action on her liquid token wallet, but it fails and Alice loses all the farming rewards.

Recommendations

Short term, allow only users with higher liquid token balance than their `storage::locked_asset_balance` to unfarm other users. Consider making the functionality for unfarming other users a part of the `seize` action flow.

Long term, consider malicious behavior while designing the protocol's authentication and authorization system. Document all the actors, their roles and responsibilities, and their restrictions to discover security vulnerabilities.

8. Users can prevent their locked assets in the STON.fi liquid token wallet from being seized

Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-FACT-8
Target: `contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func`	

Description

Users are able to prevent their locked assets in their STON.fi liquid token wallet from being seized by temporarily setting the `locked_asset_cash` state variable to zero, causing the `seize_internal` operation to fail.

A user whose liquid token wallet balance is smaller than their underlying jetton balance (`locked_asset_balance`) can have their underlying balance taken by another user whose liquid token balance is larger than their own underlying balance. This mechanism is intended to enable liquidation of the underlying balance when the liquid token is used as collateral in the Affluent lending pool.

The `seize` operation will take at least four blocks until the locked balances of the user are adjusted in the `seize_internal` operation if the user's liquid token wallet is connected to the pool account. A snippet of the `seize_internal` operation is shown in figure 8.1:

```
if (payload::op == op::seize_internal) {
  (int amount, slice seizer_address, slice seizer_wallet_address) =
(in_msg_body~load_coins(), in_msg_body~load_msg_addr(),
in_msg_body~load_msg_addr());
  try {
    throw_unless(error::enough_locked_asset_balance, storage::locked_asset_balance -
total_balance >= amount);
    throw_unless(error::not_enough_locked_asset_cash, storage::locked_asset_cash >=
amount);

    storage::locked_asset_balance -= amount;
    storage::locked_asset_cash -= amount;

    builder forward_payload = begin_cell().store_op(op::seize_success);
    send::jetton_wallet::tansfer(
      storage::asset_wallet_address,
      seizer_address,
      seizer_address,
      amount,
      100000000,
      forward_payload
    );

    save_data();
    return ();
  } catch(_, _) {

    send::liquid_farm::seize_fail(seizer_wallet_address, amount);
    return ();
  }
}
```

Figure 8.1: A snippet of the `seize_internal` operation
(`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L267–L293`)

Figure 8.1 shows that the `seize_internal` operation will fail if the `locked_asset_cash` storage variable is smaller than the provided `amount`.

However, the `sync_locked_asset_cash` operation takes only one block to temporarily reduce the storage value of `locked_asset_cash` to zero, as shown in figure 8.2:

```
if (ctx.op() == op::sync_locked_asset_cash) {
  int expect_cash = in_msg_body~load_coins();

  throw_unless(error::sync_in_process, storage::locked_asset_cash_before_sync == 0);
  throw_unless(error::invalid_expect_cash, expect_cash >
storage::locked_asset_cash);

  send::jetton_wallet::tansfer(
    storage::asset_wallet_address,
    my_address(),
    ctx.sender_address(),
    expect_cash,
    100000000,
    begin_cell()
  );

  storage::locked_asset_cash_before_sync = storage::locked_asset_cash;
  storage::locked_asset_cash = 0;
  save_data();
  return ();
}
```

Figure 8.2: A snippet of the `sync_locked_asset_cash` operation
(`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L159–L178`)

This allows a user to prevent another user from seizing their assets simply by executing the `sync_locked_asset_cash` operation one block before the `seize_internal` operation is to be executed. This way, they can continuously prevent a liquidator from seizing their assets and disincentivize liquidators from liquidating positions that use the STON.fi liquid token as collateral.

Exploit Scenario

Eve mints 100 liquid tokens for 100 underlying jettons and deposits the liquid tokens as collateral. She transfers another 100 underlying jettons to her liquid token wallet but does not trigger a transfer notification, making her `locked_asset_cash` lower than the actual amount that her liquid wallet holds. After some time, Eve's position becomes liquidateable and Alice, a liquidator, attempts to liquidate her position and `seize` her underlying assets. Eve notices this and executes the `sync_locked_asset_cash` operation one block before the `seize_internal` operation initiated by Alice is executed. This sets Eve's `locked_asset_cash` storage value to zero, causing the `seize_internal` message to fail with an exception. Eve can do this any time someone attempts to seize her underlying assets, as long as her actual balance of underlying assets is larger than the `locked_asset_cash` storage variable, essentially preventing the liquidation of her assets for an indefinite amount of time and causing the liquidators to waste gas.

Recommendations

Short term, prevent the `sync_locked_asset_cash` operation from being executed if a `seize` operation is in progress. This can be done by adding a Boolean state variable that is flipped when the first `seize_internal` operation is triggered, before the `provide_account_status` message is sent.

Long term, carefully analyze race conditions among different user actions and message flows to discover and prevent unexpected protocol behavior.

9. Vault contract can become frozen due to incorrect balance reserve amount	
Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-FACT-9
Target: `contracts/vault/share_vault/share_vault.fc`	

Description

The `Vault` contract reserves the incorrect amount of its TON balance while sending excess gas back to the user after processing user `transfer_notification` messages. As a result, the smart contract may become frozen.

The TON blockchain has been configured to freeze smart contracts if their storage rent due becomes more than `1e8` nanotons. When a smart contract is frozen, its code and data are deleted from the blockchain storage, and a 32-byte hash of the code and data is stored. This hash value allows users to recreate the account with the same code and data as it had when it was frozen. All user transactions sent to a frozen smart contract fail with an exception.

The `Vault` contract sends back excess gas to the user after processing their message by calling the `send_excesses` function, which reserves an amount of TON for the Vault contract's balance and sends the remaining to the user.

However, multiple calls to the `send_excesses` function specify the wrong reserve amount. They specify the minimum of either the `ctx::ton_balance_before_msg` value or `1e8` nanotons instead of the maximum:

```
send_excesses(query_id, response_address, min(ctx::ton_balance_before_msg,
NANOTON_1e8));
```

Figure 9.1: `contracts/vault/share_vault/share_vault.fc#L57`

The following `send_excesses` function invocations reserve the wrong amount:

1. `contracts/vault/share_vault/share_vault.fc#L57`
2. `contracts/vault/share_vault/share_vault.fc#L96`
3. `contracts/vault/share_vault/share_vault.fc#L130`

Reserving a lower amount may cause the Vault contract to have a balance of less than `1e8` nanotons. After some time of inactivity on the Vault contract, the storage fee due can become more than `1e8` nanotons, which would cause the Vault contract to be frozen. As a result, all user funds will be stuck in the Vault contract. The Affluent team can recreate the contract by sending a message with an init state data with the same hash stored in the contract.

Exploit Scenario

While processing a `transfer_notification` message, the Vault contract's initial TON balance was `1e6` nanotons. After processing the message, the Vault contract reserves `1e6` nanotons and sends the remaining TON to the `response_address`. After a couple of months of inactivity, the Vault contract is frozen.

Recommendations

Short term, fix all incorrect `send_excesses` calls so that they reserve the maximum of either the `ctx::ton_balance_before_msg` value or `1e8` nanotons while sending excess TON to the user.

Long term, improve the test suite to include test cases that check smart contract TON balance after user actions to ensure that users cannot drain a contract's TON balance.

10. Rounding down in the repay operations allows users to repay less than they owe	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-10
Target: `contracts/core/pool/handler.fc`, `contracts/core/account/handler.fc`	

Description

When an interest accrual happens in the middle of a `repay` operation, the calculation of leftover shares that need to be repaid can round down to zero and allow the user to repay a smaller amount of assets than required while clearing their entire debt.

The `repay` operation can be initiated by transferring the jetton amount to the pool and including the `repay` request in the forward payload. The pool will fetch the oracle price and then send a message containing the pool information used for calculating the repay shares along with the `execute_action(op::repay)` operation to the user's `Account` contract.

```
;; Calculate repay share.
int repay_share = repay_amount.to_borrow_share(asset);

(_, int borrowed_share, _) = status.get_account_status?(asset_address);

;; Calculate real repay amount.
int real_repay_share = min(repay_share, borrowed_share);

cell status_delta = new_account_status_delta().add_borrow_delta(asset_address, -1 *
real_repay_share);
status~apply_status_delta(status_delta);
save_data();

send::execute_action(query_id, msg::success_action(real_repay_share), action_cs);
return ();
```

Figure 10.1: A snippet of the `handler::repay` function in the account handler (`contracts/core/account/handler.fc#L114–L127`)

Next, a message is executed in the `Pool` contract that recalculates the repay amount and either refunds the excess amount or reduces the user's repaid shares and amount based on the leftover amount.

```
;; Accrue interest.
;; Make sure the interest is accrued before the all actions.
asset~accrue_interest(pool::protocol_fee_rate);

int refund_amount = 0;
int repay_amount = repay_share.to_borrow_amount(asset);
if (repay_amount < in_asset_amount) {
    refund_amount = in_asset_amount - repay_amount;
    send_jetton_transfer_msg(query_id, asset.get_wallet(), actor_address,
response_address, refund_amount, jetton_forward_ton_amount, jetton_forward_payload);
} elseif (repay_amount > in_asset_amount) {
    var borrow_back_share = (repay_amount - in_asset_amount).to_borrow_share(asset);
    repay_share -= borrow_back_share;
    repay_amount = in_asset_amount;
    cell status_delta = new_account_status_delta().add_borrow_delta(asset_address,
borrow_back_share);
    send_update_account_status_msg(query_id, recipient_address, status_delta,
response_address);
}

;; Update the pool status.
asset~decrease_total_borrow_amount(repay_amount);
asset~decrease_total_borrow_share(repay_share);
asset~increase_cash(repay_amount);
```

Figure 10.2: A snippet of the `handler::repay` function in the pool handler (`contracts/core/pool/handler.fc#L163–L183`)

However, if additional interest was accrued so that the value of `repay_amount` is larger than `in_asset_amount`, but the difference between the two rounds down to zero in the `to_borrow_share` function, the user is able to repay their entire borrow amount and shares without actually depositing this difference. This allows a user to repay less than they owe, up to the amount of interest that was accrued between the execution of the `handler::repay` function in the `Account` contract and the `handler::repay` function in the `Pool` contract (per repay), but still clear their entire debt. The leftover debt is instead distributed to the remaining users in the lending pool.

Exploit Scenario

Let us assume that a lending pool is empty and Eve deposits some amount of collateral to be able to borrow 10^9 amount of assets. She initiates the repay message sequence with a repay amount of $10^9 - 1$. The calculations performed are described below:

1. In the Account contract's `handler::repay` function, `repay_shares` is calculated as follows:

$$total\_borrow\_share \ * \ amount \ \div \ total\_borrow\_amount$$

$$= 10^9 * (10^9 - 1) \ \div 10^9 = 10^9 - 1$$

2. In the Pool contract's `handler::repay` function, `repay_amount` is calculated as follows:

$$repay\_shares \ * \ total\_borrow\_amount \ \div \ total\_borrow\_share$$

$$= \ (10^9 - 1) \ * \ (10^9 + \ 1) \ \div \ 10^9 \ = \ 10^9$$

Since `repay_amount` is greater than `in_asset_amount` $(10^9 > (10^9 - 1))$, this delta is used to calculate the remaining shares that will not be repaid; however, this calculation truncates the result down to zero:

$$borrow\_back\_share \ = \ 1 \ * \ 10^9 \div (10^9 + 1) \ = \ 0$$

Eve has cleared her debt but avoided repaying 1 nanoton of assets. While this example uses a 1 nanoton loss for the sake of simplicity, depending on the amount of interest accrued and the ratio between the value of `total_borrow_amount` and `total_borrow_share`, this amount could be significantly larger.

Recommendations

Short term, modify the `borrow_back_share` calculation so that it rounds up the result.

Long term, carefully analyze the rounding directions for all arithmetic operations and ensure that they round in favor of the protocol and not the user. Follow the rounding recommendations described in appendix D.

11. Protocol fees can be set higher than 100% and transferred to any address	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACT-11
Target: `contracts/core/pool/handler_fee_configurer.fc`	

Description

The fee configurer role has the ability to set the fee percentage to be collected by the protocol and to send the accrued jettons for the fee to another address.

The protocol fee value is not bounded, so any value in range can be set, as mentioned in issue TOB-FACT-2. Given that the percent type used by Affluent is a 14-bit unsigned integer that is divided by `10000` to get the actual fee value, it is possible for the percent type to have values up to `16383`, representing 163.83%.

The fee protocol is calculated as a percentage of the earned interest for the asset. If the fee collector sets the protocol fee to 100% before calling `accrue_interest` or `collect_fee`, all interest earned between that call and the last call to `accrue_interest` will be counted as the protocol fee, leaving the pool depositors without any profit.

Additionally, if the protocol fee is set to a value higher than 100%, the calculation for the new total supply, shown in figure 11.1, results in a negative number (as the protocol fee will be greater than the interest), decreasing the total supply instead of increasing it.

```
asset~increase_total_borrow_amount(interest);
asset~increase_total_supply_amount(interest - protocol_fee);
asset~increase_collected_protocol_fee(protocol_fee);
```

Figure 11.1: `contracts/core/pool/asset.fc#L245–L247`

Finally, the `collect_fee` operation receives the protocol fee recipient address as a parameter, thereby allowing the fee configurer to send the collected fees to any address.

The fee configurer role is set using an error-prone one-step operation (TOB-FACT-4), so this role could more easily be set to a malicious account that could exploit this issue.

Exploit Scenario

Charlie, the fee configurer, detects that a certain time has passed since the last call to `accrue_interest` for a given pool asset. He notices that a significant amount of interest has accrued, so he sets the protocol fee to the maximum value and sets the recipient address to his own personal address. User funds are impacted because they get no interest on their investment and because Charlie took assets from the pool.

Recommendations

Short term, fix issues TOB-FACT-2 and TOB-FACT-4, which will address the unbounded fee limit and role ownership configuration parts of this issue. Make the fee recipient address constant per asset or implement a whitelist for recipient addresses to prevent the protocol fees from being sent to an incorrect address.

Long term, document how the fee configurer account will be protected and under which circumstances the protocol fee will be modified. Use a timelock contract for modifying the protocol fee so that the fee cannot be directly and immediately modified and that users can act before any such modification is enforced.

12. Hard-coded forward payload for jetton transfers from the Vault contract	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FACT-12
Target: `contracts/vault/share_vault/share_vault.fc`, `contracts/vault/share_vault/imports/handler.fc`	

Description

The `Vault` contract sets the `forward_payload` of all jetton transfer messages to the unwrap wrapped TON (WTON) operation, even messages for operations that do not unwrap WTON.

```
send_jetton_transfer_msg(query_id, 0, CARRY_REMAINING_GAS, 20000000, sender_address,
from_address, from_address, amount, msg::wton_unwrap_payload());
```

Figure 12.1: `contracts/vault/share_vault/share_vault.fc#L71`

The following jetton transfer messages are affected by this issue:

1. `contracts/vault/share_vault/share_vault.fc#L71`
2. `contracts/vault/share_vault/share_vault.fc#L125`
3. `contracts/vault/share_vault/imports/handler.fc#L11`
4. `contracts/vault/share_vault/imports/handler.fc#L123`

If the unwrap WTON operation is set as the `forward_payload` for jetton transfers that do not involve this operation, the processing of the `transfer_notification` message sent to the jetton receiver will result in an exception.

Recommendations

Short term, add a check to ensure that the unwrap WTON operation is added as the `forward_payload` only to messages involving that operation and not other jetton transfer messages.

Long term, expand the test suite to test operations involving multiple jettons with different functionalities to test protocol behavior when such operations occur.

13. Liquidators can seize non-collateral assets from lending pools

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-13
Target: `contracts/core/pool.fc`	

Description

When a borrower's risk ratio or leverage ratio becomes more than the configured value, liquidators can repay the loan and seize non-collateral assets from the borrower's deposits. This allows liquidators to profit by seizing an appreciating asset when the collateral asset depreciates. The borrowers would unexpectedly lose all the unrealized profits on their deposited assets along with the loss on the depreciating collateral asset.

```
throw_unless(error::invalid_asset, seize_asset_found?);
throw_if(error::seize_asset_is_repay_asset, equal_slices(seize_asset_address,
asset_address));
```

Figure 13.1: `contracts/core/pool.fc#L132–L133`

Recommendations

Short term, take one of the following actions:

- Add a check to ensure that only collateral assets can be seized by liquidators.
- Document this behavior to warn borrowers that their non-collateral assets may be seized.

Long term, consider unexpected market movements while designing the protocol features. Document new features and non-conventional protocol behavior to make users aware of potential risks and benefits of such features.

14. Vault managers can collude with pool owners to get interest-free loans	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-14
Target: `contracts/vault/share_vault/imports/handler_manager.fc`	

Description
A vault manager can change the target weight of any whitelisted pool. Additionally, a vault manager can withdraw assets from any whitelisted pool and deposit these assets into other whitelisted pools. These functionalities help the vault manager maintain a healthy asset allocation among all whitelisted pools.

However, a vault manager can collude with a pool owner to set the pool IRM configuration to make the interest rate zero. Then, the vault manager can divert all user funds in the vault to this zero interest rate pool. This would allow the pool owner and the vault manager to borrow large amounts of assets interest-free to generate yield from other pools and protocols. As a result, the vault depositors would lose on the interest income from their deposited assets.

Note that the Affluent team will initially create and manage all the pools and vaults. Additionally, the Affluent team plans to allow third parties to create and manage pools only after making them immutable. These limitations reduce the risk of collusion to harm the protocol users.

Recommendations
Short term, take one of the following actions:

- Implement a timelock mechanism for the vault manager's `set_target_weight` and `withdraw_from_pool` actions. This will allow users time to withdraw their assets before the vault manager can misuse them.
- Document this behavior to inform borrowers of risks associated with third-party pools and vaults.

Long term, consider the risk of collusion among different system actors while designing the incentive mechanism and access control system.

15. Multiple supply and withdrawal requests from the same user to a Vault contract will fail	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-15
Target: `contracts/vault/share_vault/pool_aggregator.fc`	

Description

Multiple supply or withdrawal requests sent by a user to a `Vault` contract before the first one completes will fail because of the limitation of having a single Pool Aggregator contract per user per asset per `Vault` contract.

The `Vault` contract deploys a Pool Aggregator contract to collect the latest state of all the whitelisted pools. The Pool Aggregator contract's address depends on the `Vault` contract address, user account address, and asset address. A redeployment of the Pool Aggregator contract fails and sends an empty pool data dictionary to the `Vault` contract because of the check shown in the figure below:

```
if (op == op::provide_aggregated_pool) {
    throw_unless(error::invalid_address, equal_slices(sender_address,
vault_address));
    try {
        throw_unless(error::already_in_process, aggregated_pools.dict_empty?());
        aggregated_pools = in_msg_body~load_dict();
    ...
```

Figure 15.1: `contracts/vault/share_vault/pool_aggregator.fc#L32–L36`

Once the Pool Aggregator contract collects the pool state data from all the pools, it sends this data to the `Vault` contract and destroys itself. A new supply or withdrawal request from the user deploys a new Pool Aggregator contract to the same address.

Only a single Pool Aggregator contract can be deployed for a user, asset, and `Vault` contract at a time; therefore, any new supply or withdrawal requests will fail while a supply or withdrawal request is waiting for the Pool Aggregator contract to return the pool data. Users submitting such requests would lose gas and the third-party integrating contracts would exhibit unexpected behavior.

Recommendations

Short term, take one of the following actions:

- Implement a mechanism for the Pool Aggregator contract to collect multiple supply and withdrawal requests while it waits for pool state data from all the pools.
- Document this limitation of the `Vault` contract to inform users and third-party integrators.

Long term, document limitations of the protocol contracts to make users aware of the potential risks associated with using the protocol.

16. sync_locked_asset_cash can be executed multiple times	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-16
Target: `contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func`	

Description

Due to an ineffective check, the `sync_locked_asset_cash` operation can be called again even if the transfer notification for the first call to this operation has not yet been executed.

The `sync_locked_asset_cash` operation is intended to sync the `locked_asset_cash` storage variable by transferring the asset amount to the liquid jetton wallet itself. At the end of the message sequence, the transfer notification will update this storage variable to the `amount` provided to the `sync_locked_asset_cash` operation:

```
if (equal_slice_bits(from_address, my_address())) {
  storage::locked_asset_cash = amount;
  storage::locked_asset_cash_before_sync = 0;
  save_data();
  return ();
}
```

Figure 16.1: A snippet of the `transfer_notification` message handler in the STON.fi liquid token jetton wallet contract (`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L71–L76`)

In order to prevent multiple calls to the `sync_locked_asset_cash` operation while a call is currently in progress, this operation implements the check highlighted in figure 16.2:

```
throw_unless(error::sync_in_process, storage::locked_asset_cash_before_sync == 0);
throw_unless(error::invalid_expect_cash, expect_cash > storage::locked_asset_cash);

send::jetton_wallet::tansfer(
  storage::asset_wallet_address,
  my_address(),
  ctx.sender_address(),
  expect_cash,
  100000000,
  begin_cell()
);

storage::locked_asset_cash_before_sync = storage::locked_asset_cash;
storage::locked_asset_cash = 0;
```

```
save_data();
return ();
```

Figure 16.2: A snippet of `op::sync_locked_asset_cash` in the STON.fi liquid token jetton wallet contract (`contracts/liquid_token/liquid_farm_stonfi/jetton-wallet.func#L162–L177`)

However, if the `locked_asset_cash` storage variable is already zero, then `locked_asset_cash_before_sync` will also be set to zero. This allows anyone to call this operation multiple times.

Additionally, the `sync_locked_asset_cash` operation has a race condition with the `transfer_notification` operation. If the call to `transfer_notification` for a deposit of the underlying asset into the liquid token wallet is executed right before the call to `transfer_notification` for the `sync_locked_asset_cash` operation, the `locked_asset_cash` storage variable will disregard the newly deposited amount.

Exploit Scenario

Eve's liquid token jetton wallet has a balance of 100 underlying assets, but the `locked_asset_cash` storage variable is out of sync and set to zero. Eve executes the following transactions:

Block	First Message Sequence	Second Message Sequence
1	[liquid token wallet] sync_locked_asset_cash(amount: 100)	
2	[asset jetton wallet] op::transfer(100) balance -100	[liquid token wallet] sync_locked_asset_cash(amount: 100)
3	[asset jetton wallet] op::internal_transfer(100) balance +100	[asset jetton wallet] op::transfer(100) balance -100

4	[liquid token wallet]
	op::transfer_notification(100)
	
	locked_asset_cash = 100
		[asset jetton wallet]
		op::internal_transfer(100)
		
		balance +100

5	[liquid token wallet]	[liquid token wallet]

	op::farm
	
	locked_asset_cash = 0
		op::transfer_notification(100)
		
		locked_asset_cash = 100

If, for any reason, the first column operations are executed before the second column operations in blocks 3 and 5, Eve's `locked_asset_cash` storage variable will be larger than the available assets in her liquid token wallet.

Recommendations

Short term, use a Boolean storage variable to check if a `sync_locked_asset_cash` operation is already in progress.

Long term, improve the testing suite by testing each function with various values, explicitly using edge values such as zero. Analyze the message sequences in the protocol and determine all possible race conditions between functions that interact or share a set of important storage variables.

17. Missing or incorrect gas checks

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACT-17
Target: `contracts/liquid_token/common/liquid_farm/handler/liquid_farm.fc`, `contracts/core/pool.fc`	

Description

We found two instances of missing or incorrect gas checks in the smart contract operations. This can cause the message sequences to unexpectedly fail and could be used to intentionally bounce messages.

This affects the following operations:

- The `op::withdraw_jettons` operation in the `liquid_farm` handler of the liquid token contract does not validate the provided gas (`contracts/liquid_token/common/liquid_farm/handler/liquid_farm.fc#L66–L85`).
- The `throw_if_gas_is_not_enough` (`contracts/core/pool/gas.fc`) function used in the `Pool` contract calculates the gas fee required based on the original message forwarding fee and the number of expected messages within the particular operation's message sequence, among other parameters. Since the pool adds the summary of the asset dictionary from the pool contract storage to the account contract messages, this gas cost calculation will underestimate the message forwarding fee and lead to incomplete message sequences, resulting in an inconsistent system state.

Recommendations

Short term, analyze the gas costs for the pool operations and ensure the amount calculated in the `throw_if_gas_is_not_enough` function is enough to finalize the transaction sequence. Consider adding a gas check to the `withdraw_jettons` operation.

Long term, create a way to quickly analyze the gas costs required to execute the protocol message sequences and run this analysis each time a major code change occurs.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity, 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
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 exceeded 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.

- **Standardize the handling of bounced messages.** Contracts have different ways of checking for bounce flags, which can be confusing. Here are some examples:

```
if (ctx.bounced?()) {
    return ();
}
```

```
if (is_bounced?(flags)) {
    return ();
}
```

```
if (flags & 1) {
    return ();
}
```

- **Fix the spelling error** in the name of the `send::jetton_wallet::tansfer` functions in the DeDust and STON.fi liquid farm jetton contracts.
- **Remove unused variables in account storage.**
- **Do not use a hard-coded constant for the error code in the RedStone oracle.**
- **Standardize the codebase's naming conventions.**
  - Names that refer to the same value or concept should match. For example, `liquidate_close_ratio` and `liquidation_close_ratio` refer to the same concept but have slightly different names.
  - Some contracts add prefixes for storage variables (for example, `redstone_onchain_oracle`), and some do not (for example, `pool_aggregator`). The ones using prefixes are not consistent (for example, `share_vault`).
  - Similar to the previous point, functions that send messages do not have a clear naming convention. Some contracts use the `send::` prefix, and some use the `send_` prefix.

- **Rewrite the overriding initialize operation handler in the DeDust jetton wallet.** In the current code, the check for `op::initialize` is executed in both branches.
- **Avoid code repetition using functions.** For example, the pool contract checks repeatedly for assets.

D. Rounding Recommendations

The protocol's arithmetic rounds down on every operation, which can lead to a loss of precision that benefits the user instead of the system. This was the root cause of issues TOB-FACT-3 and TOB-FACT-10.

The following guidelines describe how to determine the rounding direction per operation. We recommend applying the same analysis for all arithmetic operations.

Determining Rounding Direction

Consider the expected output result to determine how to apply rounding (up or down).

For example, consider the formula for calculating the interest:

```
interest = time_delta * interest_rate * total_borrow_amount /
interest_rate_decimals;
```

To benefit the protocol, the rounding direction must tend toward higher values to maximize the interest accrual for the users. Therefore, it should round up.

Similar rounding techniques can be applied to all the system's formulas to ensure that rounding always occurs in the direction that benefits the protocol.

Rounding Rules

- When funds leave the system, these values should round down to favor the protocol over the user. Rounding these values up can allow attackers to profit from the rounding direction by receiving more than intended on fund interactions.

- When funds enter the system, these values should always round up to maximize the number of tokens the protocol receives. Rounding these values down can result in near-zero values, allowing attackers to profit from the rounding direction by receiving heavily discounted funds. Rounding down to zero can also allow attackers to steal funds.

- If the result of the computation can be positive or negative, then the rounding direction must mirror the sign of the result to benefit the protocol.

Recommendations for Further Investigation

- Create unit tests to highlight the result of the precision loss. Unit tests will help validate the manual analysis.

E. 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 February 10 to February 12, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Affluent team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 17 issues described in this report, Affluent has resolved 10 issues, has partially resolved two issues, and has not resolved the remaining five issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Risk of unexpected borrow amount because of race condition	Partially Resolved
2	Lack of validation for configuration values	Resolved
3	Risk of loss of fees and interest due to incorrect rounding direction in accrue_interest	Unresolved
4	Lack of two-step process for critical role transfers	Unresolved
5	The vault manager can deposit user assets into a non-whitelisted pool	Resolved
6	A malicious farming contract can be used to mint infinite liquid tokens	Resolved
7	Anyone can unfarm assets in a user’s STON.fi liquid token wallet	Unresolved
8	Users can prevent their locked assets in the STON.fi liquid token wallet from being seized	Partially Resolved
9	Vault contract can become frozen due to incorrect balance reserve amount	Resolved
10	Rounding down in the repay operations allows users to repay less than they owe	Resolved

11	Protocol fees can be set higher than 100% and transferred to any address	Resolved
12	Hard-coded forward payload for jetton transfers from the Vault contract	Resolved
13	Liquidators can seize non-collateral assets from lending pools	Resolved
14	Vault managers can collude with pool owners to get interest-free loans	Unresolved
15	Multiple supply and withdrawal requests from the same user to a Vault contract will fail	Resolved
16	sync_locked_asset_cash can be executed multiple times	Resolved
17	Missing or incorrect gas checks	Unresolved

Detailed Fix Review Results

TOB-FACT-1: Risk of unexpected borrow amount because of race condition
Partially resolved in commit `f764da4`. The `Pool` contract's withdraw and borrow handler functions now transfer the asset amount specified by the user instead of computing the amount from the share value returned by the `Account` contract. However, when users specify a share value for borrow or withdraw actions, they can still get an unexpected amount because of the interest accrual race condition.

TOB-FACT-2: Lack of validation for configuration values
Resolved in commit `b46d79d`, commit `201e23b`, and commit `e9b6157`. The Affluent team implemented validation logic for all the configuration parameters.

TOB-FACT-3: Risk of loss of fees and interest due to incorrect rounding direction in accrue_interest
Unresolved. The client provided the following context for this finding's fix status:

> *Thanks for checking deeply. Our dev team also examined this issue by looking at TON, USDT, and BTC prices, and we concluded that the transaction cost would far outweigh any potential gains, so calling it every block wouldn't be a viable attack vector. And even if decimal values are truncated during the normal accrue process. We think that unless over hundreds of accrues per day for each asset in each pool, the overall impact is negligible.*

TOB-FACT-4: Lack of two-step process for critical role transfers
Unresolved. The client provided the following context for this finding's fix status:

> *I understand that this refers to a structure like Ownable2Step, where the new owner must explicitly accept ownership to prevent simple mistakes. We pursue an immutable contract. But during the early stages of the project, we designed it to allow modifications for stable operation until sufficient testing is complete. Currently, the pool's owner is a multi-sig wallet equipped with a timelock feature. In the future, we plan to drop ownership entirely (set to 0x0). The timelock feature involves multiple steps such as propose-approve-(after timelock)execute, which reduces the risk of mistakes like 2 step ownership.*

TOB-FACT-5: The vault manager can deposit user assets into a non-whitelisted pool
Resolved in commit `7e3e748`. The `supply_to_pool` action handler now correctly checks the whitelist status of the manager-provided pool.

TOB-FACT-6: A malicious farming contract can be used to mint infinite liquid tokens
Resolved in commit `39bee02`. The STON.fi liquid farm jetton wallet contract no longer mints liquid tokens from the `unfarm` operation.

TOB-FACT-7: Anyone can unfarm assets in a user's STON.fi liquid token wallet

Unresolved. The client provided the following context for this finding's fix status:

> *This is a desired feature. The unfarm operation must precede the seize process. In the case of liquidation (when the balance is less than locked_asset_balance), we intentionally do not perform a permission check to allow the liquidator to execute the unfarm operation.*

TOB-FACT-8: Users can prevent their locked assets in the STON.fi liquid token wallet from being seized
Partially resolved in commit `0651b2c`. The sync operation has been updated to sync to the `locked_asset_balance` value instead of a user-specified value. A check to ensure a mismatch of `locked_asset_cash` and `locked_asset_balance` values has also been implemented to prevent users from calling sync repeatedly. However, though users can still prevent their assets from being seized, they can do it only once and cannot prevent repeated attempts.

TOB-FACT-9: Vault contract can become frozen due to incorrect balance reserve amount
Resolved in commit `9ab473a`. The `Vault` contract functions now reserve a maximum of either `1e8` nanotons or the contract's TON balance before receiving the message.

TOB-FACT-10: Rounding down in the repay operations allows users to repay less than they owe
Resolved in commit `b4bfe5c`. The repay action handler function now rounds up the computation of the `borrow_back_shares` value.

TOB-FACT-11: Protocol fees can be set higher than 100% and transferred to any address
Resolved in commit `5a8d0e8`. The Affluent team added a validation check to ensure that the value of `protocol_fee_rate` is less than 100%.

TOB-FACT-12: Hard-coded forward payload for jetton transfers from the Vault contract
Resolved in commit `f862bc7`. The unwrap WTON forward payload is now included only for the WTON vaults.

TOB-FACT-13: Liquidators can seize non-collateral assets from lending pools
Resolved through documentation of this behavior in the user documentation.

TOB-FACT-14: Vault managers can collude with pool owners to get interest-free loans
Unresolved.

TOB-FACT-15: Multiple supply and withdrawal requests from the same user to a Vault contract will fail

Resolved in commit `e0cf606`. The Pool Aggregator contract now includes an `index` parameter in the init data to make it unique for every supply action on the `Vault` contract.

TOB-FACT-16: sync_locked_asset_cash can be executed multiple times
Resolved in commit `48ec48b`. The sync mechanism has been updated to use a Boolean value to prevent multiple simultaneous sync action executions.

TOB-FACT-17: Missing or incorrect gas checks
Unresolved. The client provided the following context for this finding's fix status:

> *It's just test config issue. If assets are more than 4, owner should set the gas config more.*

F. 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 projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter 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.