Franklin Templeton BenjiSwap Differential Review

Security Assessment (Summary Report)

April 17, 2026

Prepared for:
Franklin Templeton Digital Assets

Prepared by: **Quan Nguyen**

Table of Contents

Table of Contents 1
Project Summary 2
Executive Summary 3
Project Goals 6
Project Targets 7
Project Coverage 8
Summary of Findings 10
Detailed Findings 11
1. depositFrom and swapFrom allow the owner to pull tokens from any user with an active swap approval 11
2. Fee-on-transfer tokens allowlisted for inbound swaps become trapped in the treasury 13
3. minAmountInByPair uses a direction-independent hash, preventing per-direction minimum swap amounts 15
4. Custody-altering functions setTreasury and setTokenHeldInContract do not require the contract to be paused 17
5. Stale pendingTreasury survives custody mode migration, allowing unilateral treasury injection 19
6. swapFrom reimplements core swap logic with multiple behavioral divergences from _executeSwap 21
7. Destination override approvals are self-granted, making amount and deadline limits unenforceable 24
A. Vulnerability Categories 26
B. Fix Review Results 28
Detailed Fix Review Results 28
C. Fix Review Status Categories 31
About Trail of Bits 32
Notices and Remarks 33

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

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

The following consultant was associated with this project:

Quan Nguyen, Consultant
quan.nguyen@trailofbits.com

Project Timeline

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

Date	Event
March 12, 2026	Pre-project kickoff call
March 23, 2026	Delivery of report draft
March 23, 2026	Report readout meeting
April 17, 2026	Delivery of final summary report

Executive Summary

Engagement Overview

Franklin Templeton engaged Trail of Bits to review the security of updates to its BenjiSwap smart contract. This re-engagement focused on changes made to `SwapPool.sol` since our October 2025 assessment, along with newly extracted libraries and interfaces. The primary additions include a hybrid custody model allowing per-token configuration of contract-held versus external treasury custody, delegated swap and deposit operations (`swapFrom`, `depositFrom`) for institutional custodians, fee-on-transfer token support with balance-delta accounting, and remediation of the two findings from the prior engagement.

One consultant conducted the review from March 16 to March 20, 2026, for a total of one engineer-week of effort. Our testing efforts focused on the custody transition logic, delegated operation authorization, treasury rotation flows, and the consistency of parallel swap code paths. With full access to source code, documentation, and the prior audit report, we performed static and dynamic testing of the codebase, using automated and manual processes. The multisig contracts were out of scope; our analysis of owner-gated functions assumes the multisig correctly enforces its authentication and threshold requirements.

Observations and Impact

The review identified seven findings spanning access controls, data validation, and code quality. The most notable finding concerns the `depositFrom` and `swapFrom` functions, which can pull tokens from any address with an active ERC-20 allowance into the `SwapPool`, regardless of whether that allowance was granted for swapping or depositing (TOB-FT-SWAPDIFF-1). A compromised owner key could use this to drain token balances from all users who have approved the pool for swaps. We also found that fee-on-transfer tokens, once allowlisted and swapped into the treasury, cannot be sent outbound through any standard pool operation, as all outbound paths enforce exact delivery with no corresponding bypass (TOB-FT-SWAPDIFF-2).

A recurring theme across several findings is incomplete state reconciliation during custody mode transitions. Two of the three custody-altering functions do not require the contract to be paused (TOB-FT-SWAPDIFF-4), and enabling contract-held mode does not clear a pending treasury proposal, allowing a previously proposed address to inject itself as the active treasury without admin involvement (TOB-FT-SWAPDIFF-5). These issues stem from custody-related state being distributed across multiple independent variables (`treasury`, `pendingTreasury`, `contractHeldTreasury`, `tokenHeldInContract`) without a centralized reconciliation mechanism.

The codebase benefits from a fixed 1:1 swap rate that is inherently resistant to front-running and sandwich attacks, a thorough test infrastructure encompassing 707 unit

and integration tests alongside Echidna fuzzing and Foundry invariant suites, and detailed event coverage across all state-changing operations. The library extraction effort (`ValidationLib`, `AssertionLib`, `AuthorizationLib`) improves modularity, though the `swapFrom` function was not refactored to use the shared swap logic, resulting in a duplicated pipeline with three observable behavioral divergences from the standard swap path (TOB-FT-SWAPDIFF-6).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Franklin Templeton take the following steps:

- **Remediate the findings disclosed in this report.** The access control gap in `depositFrom` and the fee-on-transfer token trapping warrant priority attention. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Refactor `swapFrom` to delegate to a shared internal function with `_executeSwap`.** The duplicated swap pipeline has already diverged in three ways and will require ongoing manual synchronization as the contract evolves. Eliminating the duplication removes a maintenance burden and a source of latent inconsistency.
- **Document and formalize the expected administrative operation sequences for custody migrations.** Several findings depend on the order in which admin functions are called. Codifying these sequences in operational runbooks, and where possible enforcing them on-chain, will reduce the risk of misordered transitions in production.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

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

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Data Validation	5

Project Goals

The engagement was scoped to provide a security assessment of updates to the Franklin Templeton BenjiSwap contract. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are access controls correctly enforced across all entrypoints, ensuring that only authorized traders can execute swaps and only the owner can perform administrative operations?
- Are funds held in the treasury protected from unauthorized withdrawal, drainage, or misrouting under all supported custody configurations?
- Do custody mode transitions preserve the integrity of token balances, and is related state correctly reconciled when switching between external and contract-held treasury modes?
- Does the decimal normalization arithmetic correctly handle all supported token decimal configurations without introducing rounding errors, overflow, or precision loss that could be exploited?
- Can ERC-20 approvals granted to the SwapPool for one purpose be repurposed for a different operation by a privileged or compromised actor?
- Are administrative recovery and migration functions correctly gated to prevent accidental loss of custodied assets?
- Does the UUPS upgrade mechanism follow safe proxy patterns, including proper storage layout management and authorization of new implementations?

Project Targets

The engagement involved reviewing and testing the following target.

BenjiSwap

Repository	N/A (source code provided in a zip file)
Version	March 16, 2026
Type	Solidity
Platform	EVM

Project Coverage

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

- Manual review of the hybrid custody model in SwapPool, including the global `contractHeldTreasury` toggle, per-token `tokenHeldInContract` overrides, and the interactions between custody mode transitions and related state such as pending treasury proposals
- Manual review of the delegated operation entry points (`swapFrom` and `depositFrom`) and their authorization models, with particular attention to behavioral divergences from the standard `swap` and `deposit` code paths
- Manual review of the fee-on-transfer token support, including balance-delta accounting in both `_executeSwap` and `swapFrom`, the `allowNonStandardToken` allowlist, and token recoverability across all custody configurations
- Manual review of the extracted libraries (`ValidationLib`, `AssertionLib`, `AuthorizationLib`) and changes to `Hashing.sol`, including the assembly-level hashing functions and memory safety annotations
- Manual review of the administrative recovery functions (`recoverContractTokens`, `recoverTreasuryTokens`) and the zero-balance assertion mechanism in `AssertionLib`
- Manual review of the UUPS upgrade mechanism and storage layout compatibility with the deprecated `eoaSentThisBlockNumber` slot

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 SwapPool relies on a trusted owner (multisig) to execute custody mode transitions, token registration, and treasury rotations in the correct order. We evaluated the on-chain guards but did not audit the off-chain operational procedures, runbooks, or multisig policies that govern how these administrative sequences are coordinated. Misordered admin operations (e.g., toggling custody before draining funds, or unregistering a token before recovering it) may produce states that the contract does not prevent.

- The multisig contracts (`contracts/multisig/`) were out of scope for this re-engagement. Our analysis of owner-gated functions assumes that the multisig correctly authenticates signers and enforces threshold requirements.

Summary of Findings

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

ID	Title	Type	Severity	Status
1	depositFrom and swapFrom allow the owner to pull tokens from any user with an active swap approval	Access Controls	Medium	Resolved
2	Fee-on-transfer tokens allowlisted for inbound swaps become trapped in the treasury	Data Validation	Medium	Resolved
3	minAmountInByPair uses a direction-independent hash, preventing per-direction minimum swap amounts	Data Validation	Low	Resolved
4	Custody-altering functions setTreasury and setTokenHeldInContract do not require the contract to be paused	Data Validation	Low	Resolved
5	Stale pendingTreasury survives custody mode migration, allowing unilateral treasury injection	Access Controls	Low	Resolved
6	swapFrom reimplements core swap logic with multiple behavioral divergences from _executeSwap	Data Validation	Informational	Resolved
7	Destination override approvals are self-granted, making amount and deadline limits unenforceable	Data Validation	Informational	Resolved

Detailed Findings

1. depositFrom and swapFrom allow the owner to pull tokens from any user with an active swap approval

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FT-SWAPDIFF-1
Target: `contracts/SwapPool.sol`	

Description

Two functions that pull tokens from third-party wallets rely solely on owner authorization with no on-chain consent from the source wallet. A compromised owner can use either path to drain tokens from any address that has granted an ERC-20 allowance to the SwapPool.

The `depositFrom` function is intended for multisig-initiated deposits, allowing the owner to pull tokens from a pre-approved funding address. However, it does not distinguish between addresses that approved the SwapPool for deposits and those that approved it for swapping. Any address with an active ERC20 allowance to the SwapPool can be specified as `from`:

```
function depositFrom(address token, address from, uint256 amount) external onlyOwner
nonReentrant {
    // ...
    IERC20(token).safeTransferFrom(from, address(this), amount);
    // ...
}
```

Figure 1.1: `depositFrom` pulls from an arbitrary `from` with no on-chain consent (`contracts/SwapPool.sol#L418-L440`)

The sibling function `deposit` prevents this with `if (from != msg.sender) revert("INVALID_DEPOSIT_SOURCE")`. No comparable safeguard exists for `depositFrom`.

The `swapFrom` path has a similar gap. Although it requires a pre-registered entry in the `swapFromAuthorizations` mapping, the `authorizeSwapFrom` function that populates this mapping is itself owner-gated and does not require consent from the source wallet:

```
function authorizeSwapFrom(
    address sourceWallet,
    address delegate,
    address destination
) external onlyOwner {
    ValidationLib.validateAddress(sourceWallet);
    ValidationLib.validateAddress(delegate);
    ValidationLib.validateAddress(destination);
    swapFromAuthorizations[sourceWallet][delegate] = destination;
    emit SwapFromAuthorized(sourceWallet, delegate, destination);
}
```

Figure 1.2: `authorizeSwapFrom` registers a third-party pull authorization without consent from `sourceWallet` (`contracts/SwapPool.sol#L587-597`)

A compromised owner can call `authorizeSwapFrom(victim, owner, attacker)` and then `swapFrom(victim, token, toToken, amount)` to pull tokens from the victim through the swap path. The authorization and the pull can execute in the same transaction, making the two-step requirement no barrier.

Both paths share the same root cause: the contract treats an ERC-20 allowance granted for swapping as sufficient authorization for an owner-initiated pull, without requiring the source wallet to explicitly consent to the specific operation.

Exploit Scenario

An attacker compromises the owner key and queries on-chain `Approval` events for the SwapPool address. For each victim with an active allowance, the attacker calls `depositFrom(token, victim, victim.balance)`, then calls `withdraw` to extract all deposited funds. The entire operation executes in a single block.

Recommendations

Short term, require demonstrable consent from the source wallet before either `depositFrom` or `authorizeSwapFrom` can pull their tokens. Two approaches are viable: require `msg.sender == sourceWallet` so the source wallet authorizes directly, or require the owner to submit a signature produced by the source wallet as proof of consent. This ensures that owner compromise alone is not sufficient to drain arbitrary wallets.

2. Fee-on-transfer tokens allowlisted for inbound swaps become trapped in the treasury

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FT-SWAPDIFF-2
Target: `contracts/SwapPool.sol`	

Description

The `allowNonStandardToken` flag permits fee-on-transfer tokens on inbound transfers but all outbound paths unconditionally enforce exact delivery, creating a one-way valve that traps these tokens in the treasury.

Inbound transfers accept the fee-adjusted amount when the token is allowlisted:

```
uint256 actualReceived = _treasuryTokenBalance(fromToken) - fromBalanceBefore;
if (actualReceived != amount) {
    if (!allowNonStandardToken[fromToken]) revert("NON_STANDARD_TOKEN_NOT_ALLOWED");
    amount = actualReceived;
}
```

Figure 2.1: Inbound fee-on-transfer adjustment in `_executeSwap` (`contracts/SwapPool.sol#L869-L873`)

Every outbound path rejects the same tokens. In `_executeSwap`, `swapFrom`, and `withdraw`, the exact-delivery check has no corresponding `allowNonStandardToken` bypass:

```
uint256 actualReceived = _treasuryTokenBalance(fromToken) - fromBalanceBefore;
if (actualReceived != amount) {
    if (!allowNonStandardToken[fromToken]) revert("NON_STANDARD_TOKEN_NOT_ALLOWED");
    amount = actualReceived;
}
```

Figure 2.2: Unconditional exact-delivery check on outbound (`contracts/SwapPool.sol#L931-L932`)

Once a fee-on-transfer token enters the treasury via a swap, it cannot leave through `swap` (reverts with `NON_STANDARD_TOKEN_OUTBOUND`) or `withdraw` (reverts with `FEE_ON_TRANSFER_OUT_NOT_SUPPORTED`). Recovery depends on the custody mode. For an EOA treasury, the EOA holder can transfer the tokens directly using their key without pool interaction. For an `ITreasury` contract treasury, `recoverTreasuryTokens` bypasses the fee-on-transfer checks entirely. For contract-held custody, the admin must first disable custody for the token (by toggling `tokenHeldInContract` or `contractHeldTreasury`)

and then call `recoverContractTokens`; this is a multi-step process that requires pausing the pool and temporarily changing the custody configuration.

Exploit Scenario

The owner allowlists a fee-on-transfer token and authorizes a pair with USDC. A user swaps the fee token for USDC; the pool accepts the fee-adjusted inbound amount and sends USDC. When another user later swaps USDC for the fee token, the outbound transfer delivers less than expected, and the transaction reverts. The fee token accumulates in the treasury with no operational way to send it out, and the authorized pair is permanently broken in one direction.

Recommendations

Short term, extend the `allowNonStandardToken` check to outbound transfers. When the outbound token is allowlisted, have the code skip the exact-delivery revert and adjust accounting to reflect the actual delivered amount. This will allow fee-on-transfer tokens to flow in both directions.

Long term, evaluate whether fee-on-transfer token support is a required feature. If it is, implement end-to-end handling that accounts for fees on both inbound and outbound legs of a swap, including adjusting the amount credited to the destination. If it is not, remove `allowNonStandardToken` entirely to eliminate the incomplete code path.

3. minAmountInByPair uses a direction-independent hash, preventing per-direction minimum swap amounts

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FT-SWAPDIFF-3
Target: `contracts/SwapPool.sol`, `contracts/utils/Hashing.sol`	

Description

The `setMinAmountIn` function computes its storage key via `_computeTokenPairHash`, which delegates to `Hashing.tokenPairHash`. This hash function canonically orders its two address arguments before hashing, making the result order-independent: `tokenPairHash(A, B) == tokenPairHash(B, A)`.

```
/// @dev Canonical pair hash — Order-independent: tokenPairHash(A,B) ==
tokenPairHash(B,A).
function tokenPairHash(address tokenA, address tokenB) internal pure returns
(bytes32 out) {
    assembly ("memory-safe") {
        let a := tokenA
        let b := tokenB
        if gt(a, b) {
            let tmp := a
            a := b
            b := tmp
        }
        let ptr := mload(0x40)
        mstore(ptr, shl(96, a))
        mstore(add(ptr, 20), shl(96, b))
        out := keccak256(ptr, 40)
        mstore(0x40, add(ptr, 64))
    }
}
```

Figure 3.1: `tokenPairHash` sorts addresses before hashing, producing the same key regardless of argument order. (`contracts/utils/Hashing.sol#L39-58`)

Because both directions resolve to the same key, calling `setMinAmountIn(USDC, FUND, X)` followed by `setMinAmountIn(FUND, USDC, Y)` silently overwrites `X` with `Y`. The owner cannot enforce distinct minimums for the A-to-B and B-to-A directions.

The same key is read in `_executeSwap` to enforce the minimum:

```
uint256 minRequired = minAmountInByPair[_computeTokenPairHash(fromToken, toToken)];
if (minRequired > 0 && amount < minRequired) revert("MIN_AMOUNT_NOT_MET");
```

Figure 3.2: The minimum is enforced against the same direction-independent key. (contracts/SwapPool.sol#L854-855)

When two paired tokens have different decimal precisions, a single minimum value expressed in native token units is economically meaningful in only one direction. For example, a minimum of `1000000` (1 USDC in six-decimal representation) is a reasonable dust filter for USDC-in swaps, but the same value applied to an 18-decimal token represents a negligible `1e-12` tokens, rendering the minimum ineffective in the opposite direction.

Exploit Scenario

The owner calls `setMinAmountIn(USDC, FUND, 1000000)` to require at least 1 USDC for USDC-to-FUND swaps. Because the hash is direction-independent, the same threshold also governs FUND-to-USDC swaps. A trader submits a FUND-to-USDC swap for `1000000` wei of FUND (a negligible `1e-12` FUND), and the minimum check passes. The swap executes despite being well below any economically meaningful threshold for the FUND-in direction, defeating the intended dust-filtering purpose of the minimum.

Conversely, the owner calls `setMinAmountIn(FUND, USDC, 1000000000000000000)` to require at least 1 FUND (18 decimals) for FUND-to-USDC swaps. The same threshold now applies to USDC-to-FUND swaps, requiring 1e18 units of USDC input, which is equivalent to one trillion USDC. This effectively blocks all USDC-to-FUND swaps entirely, as no trader can meet the minimum.

Recommendations

Short term, key the `minAmountInByPair` mapping with a direction-dependent hash. Alternatively, store two separate minimum values per pair (one for each direction) under the existing canonical key. This will allow the owner to configure distinct thresholds that are economically appropriate for each token's decimal precision.

Long term, review all mappings keyed by `_computeTokenPairHash` to determine whether they are truly direction-independent by design or whether they inadvertently collapse two distinct configurations into one. Documenting the intended directionality of each mapping will prevent similar issues from being introduced in future development.

4. Custody-altering functions setTreasury and setTokenHeldInContract do not require the contract to be paused

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FT-SWAPDIFF-4
Target: `contracts/SwapPool.sol`	

Description

The contract enforces a paused-state requirement for `setContractHeldTreasury` and `setContractHeldTreasuryWithAssertions`, recognizing that custody mode transitions during active swap traffic carry operational risk:

```
function setContractHeldTreasury(bool enabled) external onlyOwner nonReentrant {
    if (!paused()) revert("MUST_BE_PAUSED");
    ...
}
```

Figure 4.1: `setContractHeldTreasury` requires the contract to be paused before toggling custody mode (`contracts/SwapPool.sol#L1184-1186`)

Two other functions that modify custody routing do not enforce this requirement. `setTreasury` changes the external treasury address and atomically disables contract-held mode:

```
function setTreasury(address newTreasury) external onlyOwner {
    if (newTreasury == address(0)) revert("ZERO_ADDRESS");
    if (newTreasury == address(this)) revert("TREASURY_CANNOT_BE_SELF");
    if (requireContractTreasury && !ValidationLib.isContract(newTreasury))
        revert("TREASURY_MUST_BE_CONTRACT");
    address old = treasury;
    treasury = newTreasury;
    contractHeldTreasury = false;
    pendingTreasury = address(0);
    emit TreasuryUpdated(old, newTreasury);
}
```

Figure 4.2: `setTreasury` sets `contractHeldTreasury = false` and redirects all custody without a pause check. (`contracts/SwapPool.sol#L1220-1233`)

`setTokenHeldInContract` toggles per-token custody between the contract and the external treasury:

```
function setTokenHeldInContract(address token, bool heldInContract) external
onlyOwner {
```

```
    ValidationLib.validateAddress(token);
    if (contractHeldTreasury) revert("REDUNDANT_CUSTODY_OVERRIDE_GLOBAL");
    tokenHeldInContract[token] = heldInContract;
    emit TokenCustodyModeUpdated(token, heldInContract);
}
```

Figure 4.3: `setContractHeldTreasury` requires the contract to be paused before toggling custody mode. (`contracts/SwapPool.sol#L1184-1186`)

Both functions alter the routing decisions in `_treasuryTokenBalance`, `_treasurySend`, and `deposit`. When called while the pool is unpaused, an in-flight swap could read the treasury balance from one custody location (e.g., the contract) and then have its outbound transfer routed to a different one (e.g., the new external treasury), or vice versa. `setTreasury` is particularly impactful: it transitions from contract-held mode to external treasury mode in a single call, a change that `setContractHeldTreasury` requires a pause to perform.

Recommendations

Short term, add `if (!paused()) revert("MUST_BE_PAUSED")` to both `setTreasury` and `setTokenHeldInContract`. This applies the same safety invariant that `setContractHeldTreasury` already enforces: custody routing changes require the pool to be paused so that no swaps are in flight during the transition.

Long term, consolidate all custody-modifying operations behind a single internal modifier or helper that enforces the pause requirement uniformly. This prevents future custody-related functions from accidentally omitting the check.

5. Stale pendingTreasury survives custody mode migration, allowing unilateral treasury injection

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FT-SWAPDIFF-5
Target: `contracts/SwapPool.sol`	

Description

The contract has three functions that change the active treasury or custody mode: `setTreasury`, `setContractHeldTreasury`, and `setContractHeldTreasuryWithAssertions`. `setTreasury` clears `pendingTreasury` when assigning a new treasury address, preventing a stale proposal from being accepted after the owner bypasses the two-step flow. `setContractHeldTreasury` and `setContractHeldTreasuryWithAssertions` do not apply the same clearing logic; both set `treasury = address(0)` when enabling contract-held mode but leave `pendingTreasury` intact:

```
function setContractHeldTreasury(bool enabled) external onlyOwner nonReentrant {
    if (!paused()) revert("MUST_BE_PAUSED");
    if (enabled) {
        treasury = address(0);
        contractHeldTreasury = true;
        emit ContractHeldTreasuryUpdated(true);
    }
    ...
}
```

Figure 5.1: `setContractHeldTreasury` does not clear `pendingTreasury` when migrating to contract-held mode (`contracts/SwapPool.sol#L1184-1198`)

Meanwhile, `acceptTreasury` has no guard against being called while `contractHeldTreasury` is true. It writes a non-zero address to `treasury` without clearing the flag, and requires no admin involvement; only the pending address itself must call it.

This creates a three-step sequence in which a previously proposed address silently becomes the active treasury:

1. The admin calls `proposeTreasury(X)`, setting `pendingTreasury = X`.
2. Circumstances change and the admin calls `setContractHeldTreasury(true)`, setting `treasury = address(0)` and `contractHeldTreasury = true`. The pending proposal is not cleared.

3. X calls `acceptTreasury()` at any time — no admin action or approval is needed. This sets `treasury = X` while `contractHeldTreasury` remains true.

While `contractHeldTreasury` is true, the stale `treasury` address is dormant: all routing in `_treasuryTokenBalance` and `_treasurySend` short-circuits on the flag. However, if the admin later calls `setContractHeldTreasury(false)`, it checks only that `treasury != address(0)`, which passes because X was injected in step 3. The pool immediately begins routing outbound transfers through X.

Recommendations

Short term, have the code clear `pendingTreasury` in both `setContractHeldTreasury` and `setContractHeldTreasuryWithAssertions` when enabling contract-held mode, consistent with the clearing already performed in `setTreasury`. Additionally, add a guard to `acceptTreasury` that reverts when `contractHeldTreasury` is true, since accepting an external treasury address while in contract-held mode produces an inconsistent state regardless of how it was reached.

Long term, extract all treasury state mutations (`treasury`, `pendingTreasury`, `contractHeldTreasury`) into a single internal function that enforces the invariant: when any custody-affecting variable changes, all related variables are reconciled atomically. This prevents future custody functions from inheriting the same class of omission.

6. swapFrom reimplements core swap logic with multiple behavioral divergences from _executeSwap

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-FT-SWAPDIFF-6
Target: `contracts/SwapPool.sol`	

Description

The `swapFrom` function reimplements the full swap pipeline rather than delegating to `_executeSwap`. The two code paths diverge in at least three observable ways: a missing token-pair validation, a reordered balance-sufficiency check, and a missing destination authorization check. Each divergence weakens the invariants that `_executeSwap` enforces.

Divergence 1: Missing `validateTokenPair` call. Every public swap entrypoint calls `ValidationLib.validateTokenPair` before entering `_executeSwap`:

```
function swap(
    address fromToken,
    address toToken,
    uint256 amount
) external nonReentrant whenNotPaused onlyIfAuthorized(fromToken, toToken) {
    ValidationLib.validateTokenPair(fromToken, toToken);
    ValidationLib.validateAmount(amount);

    _executeSwap(fromToken, toToken, amount, msg.sender);
}
```

Figure 6.1: The three-argument swap validates the token pair before calling `_executeSwap` (`contracts/SwapPool.sol#L201-210`)

`swapFrom` omits this call entirely:

```
function swapFrom(
    address sourceWallet,
    address fromToken,
    address toToken,
    uint256 amount
) external onlyOwner nonReentrant whenNotPaused {
    address destination = swapFromAuthorizations[sourceWallet][msg.sender];
    if (destination == address(0)) revert("UNAUTHORIZED_SWAP_FROM");
    bytes32 pairHash = _computeTokenPairHash(fromToken, toToken);
    if (!authorizedTokenPairs[pairHash].isAuthorized)
revert("UNAUTHORIZED_TOKEN_PAIR");
    ValidationLib.validateAmount(amount);
```

Figure 6.2: `swapFrom` does not call `validateTokenPair`, so `fromToken == toToken` is not rejected (`contracts/SwapPool.sol#L268-278`)

`validateTokenPair` rejects zero addresses and identical tokens. An admin who mistakenly authorizes a self-pair would expose `swapFrom` to same-token swaps that the regular swap path would reject.

Divergence 2: Balance sufficiency checked against the pre-adjustment output amount. In `_executeSwap`, the inbound transfer occurs first, and the output amount is computed from the post-fee-on-transfer `actualReceived` value. The balance-sufficiency check runs against this adjusted amount:

```
uint256 actualReceived = _treasuryTokenBalance(fromToken) - fromBalanceBefore;
if (actualReceived != amount) {
    if (!allowNonStandardToken[fromToken]) revert("NON_STANDARD_TOKEN_NOT_ALLOWED");
    amount = actualReceived;
}

uint256 toTokenAmount = SwapMath.convertDecimals(
    amount,
    fromTokenDecimals,
    toTokenDecimals,
    true
);
if (toBalanceBefore < toTokenAmount) revert("INSUFFICIENT_TREASURY_BALANCE");
```

Figure 6.3: `_executeSwap` checks balance sufficiency against the fee-adjusted output (`contracts/SwapPool.sol#L869-882`)

In `swapFrom`, the output amount is computed and the balance is checked before the inbound transfer. If a fee-on-transfer token reduces the received amount, the output is then recomputed, but the sufficiency check is never re-evaluated against the updated value:

```
uint256 toAmount = SwapMath.convertDecimals(amount, fromDec, toDec, true);

uint256 fromBalBefore = _treasuryTokenBalance(fromToken);
uint256 toBalBefore = _treasuryTokenBalance(toToken);
if (toBalBefore < toAmount) revert("INSUFFICIENT_TREASURY_BALANCE");

// Transfer in from sourceWallet
...

// Handle non-standard tokens
uint256 actualReceived = _treasuryTokenBalance(fromToken) - fromBalBefore;
if (actualReceived != amount) {
    if (!allowNonStandardToken[fromToken]) revert("NON_STANDARD_TOKEN_NOT_ALLOWED");
    amount = actualReceived;
    toAmount = SwapMath.convertDecimals(amount, fromDec, toDec, true);
}
```

Figure 6.4: `swapFrom` checks balance sufficiency against the original, unadjusted `toAmount`. (`contracts/SwapPool.sol#L296-317`)

In practice, this ordering difference does not create an exploitable condition today because the fee adjustment always reduces `toAmount`, so the original sufficiency check is strictly more conservative. However, if the adjustment logic is ever generalized (e.g., to support rebasing tokens or positive slippage), the stale check would silently permit swaps the treasury cannot cover.

Divergence 3: Missing `enforceTraderAuthorization` check on destination. The four-argument `swap` enforces that, when trader authorization is active for a pair, the destination must be an authorized trader:

```
if (authorizedTokenPairs[tokenPairHash].enforceTraderAuthorization &&
    !authorizedTraders[tokenPairHash][destination])
    revert("UNAUTHORIZED_DESTINATION");
```

Figure 6.5: The four-argument `swap` enforces trader authorization on the destination. (`contracts/SwapPool.sol#L237-242`)

`swapFrom` does not replicate this check. If trader authorization is enforced for a pair and the destination stored in `swapFromAuthorizations` is later deauthorized as a trader, `swapFrom` will still deliver tokens to that destination.

Recommendations

Short term, refactor `swapFrom` to delegate its core logic to `_executeSwap` (or a shared internal function). This eliminates the duplicated code paths and ensures that all current and future validations are applied uniformly to both swap entrypoints.

Long term, add integration tests that exercise both `swap` and `swapFrom` against the same edge-case matrix (identical tokens, fee-on-transfer tokens, unauthorized destinations, deauthorized traders) and assert identical revert or success behavior. This will prevent future divergences from being introduced silently.

7. Destination override approvals are self-granted, making amount and deadline limits unenforceable

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FT-SWAPDIFF-7
Target: `contracts/SwapPool.sol`	

Description

The destination override approval mechanism follows an approval-then-spend pattern similar to ERC-20 `approve`/`transferFrom`. In the ERC-20 model, the approval grants a different party the right to spend on the approver's behalf, which is why the amount cap is meaningful: the spender cannot unilaterally raise it.

In `SwapPool`, however, both sides of the operation are the same actor. The trader calls `setDestinationOverrideApproval` to create the approval (keyed on `msg.sender`), and the trader calls `swap`, which internally consumes it (also keyed on `msg.sender`):

```
function setDestinationOverrideApproval(
    address fromToken,
    address toToken,
    uint256 amount,
    address destination,
    uint256 deadline
) external {
    ...
    bytes32 key = Hashing.hash4(msg.sender, fromToken, toToken, destination);
    destinationOverrideApprovals[key] = DestinationOverrideData({
        remainingAmount: amount,
        deadline: deadline
    });
    ...
}
```

Figure 7.1: The trader grants an approval to themselves. (`contracts/SwapPool.sol#L610-639`)

```
if (destination != msg.sender) {
    ...
    _consumeDestinationOverrideApproval(
        msg.sender,
        fromToken,
        toToken,
        amount,
        destination
```

```
    );
    ...
}
```

Figure 7.2: The same trader's swap consumes their own approval. (contracts/SwapPool.sol#L234-251)

Because the approver and the spender are the same address, the `remainingAmount` cap and the `deadline` provide no constraint that the trader has not already voluntarily accepted. The trader can reset both values at any time by calling `setDestinationOverrideApproval` again. This makes the mechanism equivalent to a simple Boolean flag indicating intent, with the `amount` and `deadline` fields adding bookkeeping cost but no enforcement value.

Recommendations

Short term, document that the destination override approval is a self-service intent signal for off-chain compliance monitoring, not an on-chain spending limit. This prevents integrators and auditors from treating the `remainingAmount` as a security-relevant bound.

Long term, if an enforceable cap on redirected volume is desired, separate the approver from the spender: for example, by requiring the owner or a compliance role to grant destination override approvals on behalf of traders. This restores the two-party trust model that makes approval amounts meaningful.

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**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
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. 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.

On April 3, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Franklin Templeton team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Franklin Templeton has resolved all seven issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	depositFrom and swapFrom allow the owner to pull tokens from any user with an active swap approval	Medium	Resolved
2	Fee-on-transfer tokens whitelisted for inbound swaps become trapped in the treasury	Medium	Resolved
3	minAmountInByPair uses a direction-independent hash, preventing per-direction minimum swap amounts	Low	Resolved
4	Custody-altering functions setTreasury and setTokenHeldInContract do not require the contract to be paused	Low	Resolved
5	Stale pendingTreasury survives custody mode migration, allowing unilateral treasury injection	Low	Resolved
6	swapFrom reimplements core swap logic with multiple behavioral divergences from _executeSwap	Informational	Resolved
7	Destination override approvals are self-granted, making amount and deadline limits unenforceable	Informational	Resolved

Detailed Fix Review Results

TOB-FT-SWAPDIFF-1: depositFrom and swapFrom allow the owner to pull tokens from any user with an active swap approval
Resolved. The team deprecated the `depositFrom` function. It now reverts, removing the owner-controlled arbitrary pull path. For `swapFrom`, owner-created authorization is no longer sufficient by itself: execution additionally requires a separate ERC-20 allowance from `sourceWallet` to the owner, so a standard allowance to `SwapPool` can no longer be repurposed to drain user funds.

TOB-FT-SWAPDIFF-2: Fee-on-transfer tokens whitelisted for inbound swaps become trapped in the treasury
Resolved. The team removed the `allowNonStandardToken` flag entirely and unconditionally rejected fee-on-transfer tokens on all inbound and outbound paths with exact-delivery checks (`_executeSwap`, `swapFrom`, `deposit`, `depositFrom`, `withdraw`). The deprecated storage slot is retained only for UUPS upgrade layout compatibility.

TOB-FT-SWAPDIFF-3: minAmountInByPair uses a direction-independent hash, preventing per-direction minimum swap amounts
Resolved. The team added a new `directedPairHash` function in `Hashing.sol` that preserves argument order (A->B != B->A), a new `minAmountInByDirection` mapping, and a `setMinAmountInByDirection` admin setter. Both `_executeSwap` and `swapFrom` now check the direction-dependent minimum first, falling back to the legacy direction-independent minimum. The legacy `minAmountInByPair` mapping and `setMinAmountIn` are retained for backward compatibility.

TOB-FT-SWAPDIFF-4: Custody-altering functions setTreasury and setTokenHeldInContract do not require the contract to be paused
Resolved. The team added pause checks to both `setTreasury` and `setTokenHeldInContract`, enforcing the same paused-state requirement that `setContractHeldTreasury` already applies.

TOB-FT-SWAPDIFF-5: Stale pendingTreasury survives custody mode migration, allowing unilateral treasury injection
Resolved. The team applied both short-term recommendations: `setContractHeldTreasury` and `setContractHeldTreasuryWithAssertions` now clear `pendingTreasury` when enabling contract-held mode, and `acceptTreasury` reverts with `CONTRACT_HELD_MODE_ACTIVE` when `contractHeldTreasury` is true.

TOB-FT-SWAPDIFF-6: swapFrom reimplements core swap logic with multiple behavioral divergences from _executeSwap
Resolved. The team addressed all three divergences in-place rather than refactoring to a shared function: (1) `swapFrom` now calls `validateTokenPair` to reject zero addresses and identical tokens, (2) the inbound transfer is performed before computing the output

amount and checking balance sufficiency, matching `_executeSwap`'s order, and (3) the `enforceTraderAuthorization` check on the destination is now enforced in `swapFrom`. The two code paths are now behaviorally consistent.

TOB-FT-SWAPDIFF-7: Destination override approvals are self-granted, making amount and deadline limits unenforceable

Resolved. The team implemented the short-term recommendation by adding NatSpec documentation to `setDestinationOverrideApproval` clarifying that the amount and deadline fields are a self-service intent signal for off-chain compliance monitoring, not an on-chain spending limit, and that this is by design.

C. Fix Review Status Categories

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

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

About Trail of Bits

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

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

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

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

To keep up 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.