Franklin Templeton BenjiSwap Contract

Security Assessment (Summary Report)

October 20, 2025

Prepared for:
Franklin Templeton Digital Assets

Prepared by: **Quan Nguyen**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

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
September 11, 2025	Pre-project kickoff call
September 22, 2025	Delivery of report draft
October 6, 2025	Report readout meeting
October 20, 2025	Delivery of final summary report

Project Targets

The engagement involved reviewing and testing the following target.

BenjiSwap

Repository	N/A (source code provided in a zip file)
Version	September 14, 2025
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

Franklin Templeton engaged Trail of Bits to review the security of its BenjiSwap smart contract. The target system centers on `SwapPool`, an upgradeable UUPS contract that performs fixed-rate one-to-one swaps with decimal normalization between authorized token pairs. Core features include per-pair and per-trader authorization, destination-override payouts guarded by owner-set approvals, and treasury outflows routed either through a contract treasury or an EOA via `Permit2` with optional expiry and per-token caps.

We reviewed the core swap and treasury flow in `SwapPool.sol` and the supporting `SwapMath.sol` library, focusing on authorization, treasury outflow limits, the token registration and token unsupported policies, balance accounting, and decimal normalization. The review combined manual source analysis with targeted reasoning about edge cases and invariants. This was a source-only, time-boxed assessment involving no execution of the full on-chain test suite or integration with a concrete `ITreasury` implementation. Some behaviors (e.g., treasury balance abstraction) depend on the specific treasury contract used. Components under `contracts/multisig/*` and `PrivateMultiSig.sol` were explicitly out of scope and treated as supporting governance utilities; accordingly, issues or interactions originating solely from those modules were not assessed.

One consultant conducted the review from September 15 to September 19, 2025, for a total of one engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We identified no high-severity issues, only two issues of low/informational severity: a low-severity business-logic gap allowing an EOA “per-transaction” cap to be bypassed via a router/multicall transaction (TOB-FT-SWAP-1), and missing policy checks in the `deposit` and `withdraw` functions that swap operations enforce, creating inconsistencies in token handling (TOB-FT-SWAP-2). In addition to these issues, we found code quality findings centered on consistency and clarity (appendix C). Overall, the code follows the checks-effects-interactions pattern, uses explicit reverts, and is structurally clear, but would benefit from centralizing policies and tightening documentation/formatting to reduce ambiguity and maintenance risk.

Recommendations

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The arithmetic is straightforward and guarded by explicit checks. The system enforces a maximum of 77 decimals when registering new tokens and validates that amounts passed to `Permit2` fit within `uint160`, which together provide strong protection against overflow and representation errors.	**Satisfactory**
Auditing	The codebase emits sufficient events for critical configuration changes and provides detailed events for core operations such as swaps, deposits, and withdrawals, supporting effective observability and post-incident analysis.	**Satisfactory**
Authentication / Access Controls	Access controls are solid, combining owner-only administration with per-pair and per-trader authorization. Operationally, pair authorization is directional (A-to-B authorization is distinct from B-to-A authorization), which is powerful but demands governance discipline; the `deposit` and `withdraw` functions do not include token registration and token unsupported policy checks, and treasury mode relies on contract/EOA distinctions that merit careful configuration.	**Satisfactory**
Complexity Management	While the contract implements many valuable validations, some checks are duplicated, and several concerns are combined within large functions, indicating room to reduce duplication and improve separation of concerns.	**Moderate**
Decentralization	The project is openly centralized.	**Not Applicable**

Documentation	The repository provides sufficient documentation to understand architecture, controls, and operational flows, enabling efficient review and maintenance.	**Satisfactory**
Low-Level Manipulation	Low-level code is limited to the hashing library's assembly, which is minimal and accompanied by clear comments, though it warrants careful maintenance due to inherent brittleness.	**Satisfactory**
Testing and Verification	The project includes comprehensive unit and integration tests, coverage tooling, Slither analysis, and Echidna fuzzing with invariants.	**Satisfactory**
Transaction Ordering	The implementation consistently applies the checks-effects-interactions pattern and uses `nonReentrant` guards on external functions; no transaction ordering issues were identified.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity	Status
1	EOA per-transaction cap can be bypassed via multicall	Data Validation	Low	Resolved
2	Treasury deposit and withdraw functions lack checks for token registration and token unsupported policies	Data Validation	Informational	Resolved

Detailed Findings

1. EOA per-transaction cap can be bypassed via multicall

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FT-SWAP-1
Target: `SwapPool.sol`	

Description

The EOA "per-transaction" cap is enforced per function call, not per transaction. A router/multicall transaction can split a large swap into many sub-cap calls within a single transaction, effectively bypassing the intended aggregate ceiling unless a per-block cap is configured.

In the EOA path of the `_treasurySend` function, the cap compares only the current call's `amount` to `eoaMaxPerTx[token]`. There is no transaction-scoped accumulator; the swap operation is `nonReentrant`, but sequential calls in the same transaction are allowed. If `eoaMaxPerBlock[token]` is 0, the aggregate in one transaction becomes unbounded.

```
uint256 perTxCap = eoaMaxPerTx[token];
if (perTxCap != 0 && amount > perTxCap) revert("EOA_MAX_PER_TX");
uint256 perBlockCap = eoaMaxPerBlock[token];
if (perBlockCap != 0) {
    // Reset per token if block changed
    if (eoaSentBlockOf[token] != block.number) {
        eoaSentBlockOf[token] = block.number;
        eoaSentThisBlock[token] = 0;
    }
    uint256 newCum = eoaSentThisBlock[token] + amount;
    if (newCum > perBlockCap) revert("EOA_MAX_PER_BLOCK");
    eoaSentThisBlock[token] = newCum;
}
```

Figure 1.1: The EOA cap checks in the `_treasurySend` function

An attacker can exceed the intended "per transaction" ceiling in one transaction. With no `eoaMaxPerBlock` set, there is no aggregate limit; if `eoaMaxPerBlock` is set, the attacker can still reach the full per-block cap in a single transaction.

Exploit Scenario

Alice operates a router contract. The contract owner sets `eoaMaxPerTx[USDC]` to 100,000 and leaves `eoaMaxPerBlock[USDC]` set to `0`. Alice wants to receive 500,000 USDC from the treasury in one transaction while staying under the per-call cap. Her router makes five sequential calls within one transaction, each calling `swap(from, to, 100_000)`. Every call passes the `EOA_MAX_PER_TX` check since `amount` is equal to `cap`. The treasury ends up sending 500,000 USDC to Alice in a single transaction, bypassing the intent of the per-transaction ceiling.

If later `eoaMaxPerBlock[USDC]` is set to 300,000, Alice simply makes three sequential 100,000 calls in one transaction and reaches the full 300,000 per-block limit in that single transaction.

Recommendations

Short term, add a transaction-scoped accumulator (e.g., transient storage per EIP-1153) to sum all amounts within the same transaction and enforce the cap against the aggregate.

Long term, deprecate EOA mode entirely to remove multicall/composability bypass risk, and cover all future call sites with centralized, testable enforcement.

2. Treasury deposit and withdraw functions lack checks for token registration and token unsupported policies

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-FT-SWAP-2
Target: `SwapPool.sol`	

Description

The owner-only `deposit` and `withdraw` functions move arbitrary tokens without checking token registration or token unsupported flags; specifically, they validate addresses and amounts but never check `registeredTokens[token].isRegistered` or `unsupportedTokens[token]`. Conversely, the `_executeSwap` function reverts on unregistered or unsupported tokens, creating a policy inconsistency where the treasury may hold or move tokens that cannot be swapped, and "unsupported" might be misread as a universal block on all movements.

```
function deposit(address token, address from, uint256 amount) external onlyOwner
nonReentrant {
    _validateAddress(token);
    _validateAddress(from);
    _validateAmount(amount);

    if (from != msg.sender) revert("INVALID_DEPOSIT_SOURCE");

    IERC20(token).safeTransferFrom(msg.sender, treasury, amount);

    uint256 newBalance = IERC20(token).balanceOf(treasury);
    emit TokenDeposited(token, amount, newBalance);
}
```

Figure 2.1: The `deposit` function in `SwapPool.sol`

Recommendations

Short term, enforce consistency by applying `registeredTokens[token].isRegistered` and `!unsupportedTokens[token]` in the `deposit` and `withdraw` functions. Alternatively, explicitly document that owner-only treasury moves are exempt from swap token policies and are an operational control.

Long term, add a shared helper function that is called in the `deposit`, `withdraw`, and `_executeSwap` functions when checking token validity for consistency.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

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

C. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **Duplicated check for `DESTINATION_CANNOT_BE_TREASURY`.** The `DESTINATION_CANNOT_BE_TREASURY` value is checked in both the external swap function and the `_executeSwap` function. This duplication is functionally harmless but adds gas overhead and increases the chance of future drift if one check is changed but not the other.

```
function swap(
    address fromToken,
    address toToken,
    uint256 amount,
    address destination
) external nonReentrant whenNotPaused onlyIfAuthorized(fromToken, toToken) {

    // [...]

    if (destination == treasury) revert("DESTINATION_CANNOT_BE_TREASURY");

    // [...]

    _executeSwap(fromToken, toToken, amount, destination);
}

function _executeSwap(
    address fromToken,
    address toToken,
    uint256 amount,
    address destination
) internal {
    // [...]

// Prevent routing the swap output back to the treasury in all paths
if (destination == treasury) revert("DESTINATION_CANNOT_BE_TREASURY");

    // [...]
}
```

Figure C.1: The DESTINATION_CANNOT_BE_TREASURY check in swap and _executeSwap functions

- **Redundant zero-address validation in `setPermit2`.** The `setPermit2` function checks `permit2_ != address(0)` and then calls `_validateAddress(permit2_)`. Since the call is already inside a nonzero conditional, `_validateAddress` (which only checks for zero) is redundant and adds noise/gas without strengthening safety.

```
function setPermit2(address permit2_) external onlyOwner {
    if (permit2_ != address(0)) {
        _validateAddress(permit2_);
        if (!_isContract(permit2_)) revert("PERMIT2_MUST_BE_CONTRACT");
    }
    permit2 = permit2_;
    emit Permit2Updated(permit2_);
}
```

Figure C.2: The `setPermit2` function in `SwapPool.sol`

- **Inconsistent indentation within function bodies.** Multiple statements and comments are misindented relative to their surrounding blocks, notably in `initialize` (lines 226–229) and `_executeSwap` (lines 591–592). This reduces readability, increases review friction, and can hide subtle logic changes in diffs.

- **Comment/code mismatch for deadline validation.** The inline comment states that the deadline "must be a future block number," but the check allows the current block (`deadline == block.number`) by reverting only when `deadline` is lower than `block.number`. This creates ambiguity about approval validity at the current block and can mislead readers or tests.

```
if (deadline < block.number) revert("INVALID_DEADLINE_VALUE"); // must be a
future block number
// ...
destinationOverrideApprovals[key] = DestinationOverrideData({
    remainingAmount: amount,
    deadline: deadline
});
```

Figure C.3: The deadline check in the `setDestinationOverrideApproval` function

- **Inconsistent balance source for contract treasuries.** The view path uses `ITreasury.getTokenBalance` (lines 516–523) when `treasury` is a contract, while the `deposit` and `withdraw` functions read balances directly via `IERC20.balanceOf(treasury)` for checks and event logging. If the treasury abstracts balances (internal accounting, wrappers, staking), these sources can diverge, causing misleading events and potentially incorrect balance checks.

- **Typo in comment.** A comment (line 61) has a spelling mistake ("overrride," with three "r"s), which can hinder searchability and clarity in reviews and tooling.

- **Lack of named constant and rationale for magic number for token decimals (77).** The `registerToken` function hard codes a maximum decimal limit of 77 without a named constant or documentation. This reduces readability, complicates reuse, and risks accidental drift if the bound changes elsewhere.

D. 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 October 2, 2025, 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 both issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	EOA per-transaction cap can be bypassed via multicall	**Resolved**
2	Treasury deposit and withdraw functions lack checks for token registration and token unsupported policies	**Resolved**

Detailed Fix Review Results

TOB-FT-SWAP-1: EOA per-transaction cap can be bypassed via multicall
Resolved. A transaction-scoped accumulator now enforces the cap by aggregating transfers keyed by `(token, tx.origin, block.number)`, preventing multicall bypass.

TOB-FT-SWAP-2: Treasury deposit and withdraw functions lack checks for token registration and token unsupported policies
Resolved. Deposits now reject tokens marked as unsupported. Withdrawals additionally require the token to be registered and not marked as unsupported.

E. 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

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

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.