Smart Contract Audit Report

Date of Audit: May 5, 2025
Version: v1.1
Audited by: Shahaf Antwarg
Client: Superform Labs
Repository: https://github.com/superform-xyz/v2-contracts
Frozen Hash: 312f23499aeeef0a65fbec856fffb25847e6155a
Contracts Reviewed: src/core - Focus on Adapters and Destination Executers

Report Properties

Client	Superform Labs
Version	v1.1
Contracts	SuperDestinationExecutor.sol, DebridgeAdapter.sol, SuperDestinationValidator.sol
Author	Shahaf Antwarg
Auditors	Shahaf Antwarg
Classification	Confidential

Version Info

Version	Commit	Date	Author	Description
v1.0	312f234	May 5 2025	Shahaf Antwarg	Release Candidate
v1.1	b1da8a7	May 6 2025	Shahaf Antwarg	Final Release

Deployments

Contract	Chain	Address	Date

Table of Contents

1. Introduction

Following our comprehensive review of the Superform v2 protocol smart contracts, this report outlines our systematic approach to evaluating potential security vulnerabilities, cross-chain messaging integrity, and account-based execution patterns. Our review focused on the core contracts that handle cross-chain bridging, message validation, and smart account interactions. Overall, while the protocol implements robust security practices including signature verification and merkle root validation, we identified several issues.

This audit focuses on the SuperDestinationExecutor (cross-chain execution endpoint), bridge adapters (DeBridge), validator implementations (signature and merkle proof verification), and related supporting contracts. The Superform team requested a review of these contracts to detect security vulnerabilities, cross-chain messaging weaknesses, and potential issues in their smart account-based cross-chain protocol.

1.1 About Superform

The Superform Protocol is a suite of non-upgradeable, non-custodial smart contracts that act as a central repository for yield and a router for users.

Superform builds on top of this to solve other core issues in DeFi, together forming a two-sided marketplace paving the way for efficient discovery, execution, and management of yield across chains. It is permissionless, modular, and enables intent-based transactions across chains that allow users to execute into an arbitrary number of tokens, chains, and vaults at once.

1.2 About node.security

node.security Audits is a dedicated team specializing in smart contract security and blockchain risk management. Our team leverages a combination of automated tools and manual review to evaluate contracts thoroughly. We adhere to industry best practices and continuously update our methodologies based on emerging threats and vulnerabilities. For any queries or additional information, we can be reached via Telegram.

1.3 Disclaimer

This audit represents an independent security review of the smart contract(s) provided and is not a substitute for the complete functional testing that should be performed before any software release. Although every effort has been made to identify vulnerabilities, no audit can guarantee that all potential issues have been discovered. We strongly recommend additional independent audits and a public bug bounty program to further strengthen the security posture. This report is intended solely for security evaluation purposes and should not be interpreted as investment advice.

2. Report Structure

This audit report is organized to facilitate clear and efficient navigation from the most critical findings to less significant issues. Each issue is carefully documented with its severity rating and status, ensuring that readers can quickly assess the overall security posture and prioritize remediation efforts.

2.1 Issue tags

Issues are tagged as:

- **Resolved**: The issue has been fixed.
- **Unresolved**: The issue remains open.
- **Verified**: The functionality has been reviewed and confirmed with the client.

2.2 Severity levels

Severity levels are defined as follows:

- **Critical**: Issues that may lead to direct loss of funds or severe misallocation.
- **High**: Issues that significantly disrupt contract operation or pose a high risk of exploitation.
- **Medium**: Issues that affect the operation in non-catastrophic ways.
- **Low**: Minor issues that have minimal impact.
- **Info**: Observations that do not impact functionality but provide guidance for best practices.

2.3 Vulnerability Categories

Our audit followed a structured checklist covering a wide range of potential issues, from basic coding errors to advanced DeFi attack vectors. This approach ensures that every aspect of the code is thoroughly examined. The categories we evaluated include:

Basic Coding Bugs:

- Constructor and initializer mismatches
- Overflows, underflows, and unchecked arithmetic
- Short address or parameter attacks
- Uninitialized storage pointers

Semantic Consistency Checks:

- Consistency between the code and the design documentation
- Clarity of comments and correct parameter naming
- Detection of misleading or outdated code comments

Access Control & Authorization:

- Verification of role-based restrictions
- Proper management of privileged operations to prevent unauthorized access

Reentrancy & External Calls:

- Identification of reentrancy vulnerabilities
- Ensuring adherence to the checks-effects-interactions pattern

Business Logic & Financial Flow:

- Correct implementation of deposit, withdrawal, staking, and reward distribution
- Enforcement of lock periods, daily withdrawal caps, and minimum deposit amounts
- Ensuring consistency between internal state and actual token balances

Advanced DeFi Attack Vectors:

- Front-running, flash loan exploitation, and MEV risks
- Oracle manipulation and price feed vulnerabilities

Resource Management & Gas Optimization:

- Efficiency of loops and unbounded iterations
- Removal of redundant operations
- Appropriate use of compiler optimizations and internal function visibility

External Integration & Compatibility:

- Interaction with ERC20 tokens and potential issues with deflationary or rebasing tokens
- Proper handling of external contract calls and integration with critical services

Coding Standards & Documentation:

- Consistent naming conventions and parameter usage
- Correct and clear code comments without typos
- Adherence to best practices in coding style and organization

This comprehensive checklist enabled us to systematically identify and address vulnerabilities, ensuring a detailed and thorough audit of the contracts.

3. Scope of Work

Contracts Audited:

- **SuperDestinationExecutor:** Cross-chain execution gateway managing bridged transaction validation, account creation, merkle root verification, and smart account interactions
- **Bridge Adapters:** Interface contracts for major cross-chain bridges (DeBridge, Across) that handle token movement and message passing between chains
- **SuperDestinationValidator:** Signature and merkle proof validation for cross-chain messages with ownership verification

Focus Areas:

1. **Cross-Chain Message Security:** Validity of message formats, signature verification, prevention of replay attacks
2. **Fund Validation:** Verification of received amounts, prevention of unauthorized fund movements, balance reconciliation
3. **Account Management:** Smart account creation, ownership verification, proper execution delegation
4. **Merkle Proof System:** Root uniqueness enforcement, leaf computation integrity, proof verification rigor
5. **Bridge Adapter Safety:** Message handling, token transfer sequence, adapter authorization mechanisms
6. **Execution Flow:** Hook processing, context preservation, failure handling without reverting critical state changes
7. **Gas Optimization:** Efficient bytecode handling, memory usage in cross-chain contexts, storage optimization
8. **Error Handling:** Comprehensive error conditions, safe execution patterns, proper event emissions for failed operations

4. Methodology

1. **Manual Code Review**: Line-by-line inspection of each contract, focusing on fund flows (deposit, withdraw), lock enforcement, and external calls.
2. **Automated Analysis**: Tools such as Slither, Mythril for static analysis and vulnerability detection (reentrancy patterns, uninitialized storage, etc.).
3. **Threat Modeling**: Considering ways an attacker could bypass daily caps, lock periods, or manipulate callbacks.
4. **Testing & Simulation**: Deployed in a test environment, tried scenarios for normal usage and malicious attempts.
5. **Reporting**: Consolidation of all findings by severity, with recommended fixes and references to the code.

5. Findings

Our audit of **Superform v2** focused on core cross-chain executors, adapters, and validators, uncovering several medium-severity issues: bridged-fund verification relies on the account's final balance instead of the actual transfer delta; Merkle roots are marked "used" on partial fills, blocking later valid fills; global balance checks allow one intent's transfers to satisfy another (cross-intent races).

Superform's signature and Merkle-proof systems are robust - addressing these findings will tighten fund validation, eliminate DoS-style intent races, and streamline adapter flows, further strengthening cross-chain execution safety.

Below is a summary of the issues found. Each includes a **severity rating** (Critical, High, Medium, Low, Informational), title, potential exploit scenarios, and our **recommended remediation**.

ID	**Severity**	**Title**	**Category**	**Status**
M-01	**Medium**	Early Merkle-Root Consumption on Under-Fills	State-Poisoning & Execution Ordering	Resolved
M-02	**Medium**	Bridged Token Verification Relies on Final Balance Instead of Isolated Transfer Validation	Fund Validation & Cross-Chain Intent Integrity	Verified
M-03	**Medium**	Cross-Intent Interference via Global Balance Check	Concurrency & Intent Isolation	Verified
L-01	**Low**	Missing Reentrancy Protection in the DeBridge Adapter	Reentrancy	Verified
I-01	**Info**	Non-Standard Access Control Pattern in DeBridge Adapter	Code Quality & Best Practices	Resolved
I-02	**Info**	Outdated "Across" References in DebridgeAdapter	Coding Standards & Documentation	Resolved
I-03	**Info**	Redundant Message Decoding in DeBridge Adapter	Gas Optimization & Code Efficiency	Resolved

M-01: Early Merkle-Root Consumption on Under-Fills

Severity: Medium
Category: State-Poisoning & Execution Ordering

Description:
In `SuperDestinationExecutor.processBridgedExecution`, the Merkle root is marked "used" before checking that the account's balance meets `intentAmount`. A partially-filled transfer (same root) will flip that flag and then return early when funds are still insufficient - preventing any later fill of that root from ever completing the intent.

Reproduction Steps:

1. User's `intentAmount = 100`, one Merkle root **R**.
2. First bridge transfer of 60 arrives under root **R** → `balance = 60`.
3. `processBridgedExecution` marks `usedMerkleRoots[…][R] = true`, sees `60 < 100`, and returns.
4. Second transfer of 40 arrives under same **R** → `balance = 100`.
5. Root **R** already marked, so execution reverts.

Recommendation:

- Move the "mark root used" below the balance check.
- Keep the `return` on under-fill (so the received funds stay in the account).

```
// --- Balance check (return on under-fill, preserving funds) ---
uint256 bal = tokenSent == address(0)
   ? account.balance
   : IERC20(tokenSent).balanceOf(account);
if (intentAmount != 0 && bal < intentAmount) {
   emit SuperDestinationExecutorReceivedButNotEnoughBalance(account);
   return;
}
+   // --- Now consume the root only on a full fill ---
+   if (usedMerkleRoots[account][merkleRoot]) revert MERKLE_ROOT_ALREADY_USED();
+   usedMerkleRoots[account][merkleRoot] = true;

// … continue with execution …
```

This change ensures:

- Partial fills do not "poison" the root flag.
- Funds delivered by partial fills remain available.
- Only the transfer that actually completes the intent (balance ≥ intentAmount) will mark - and then consume the Merkle root.

M-02: Bridged Token Verification Relies on Final Balance Instead of Isolated Transfer Validation

Severity: Medium
Category: Fund Validation & Cross-Chain Intent Integrity

Description:
In `SuperDestinationExecutor.sol`, the `processBridgedExecution` function validates whether the target `account` holds enough funds to satisfy the `intentAmount` by simply checking the current balance. It assumes that the adapter has already transferred the correct amount prior to execution:

```
// --- Balance Check ---
if (tokenSent == address(0)) {
   if (intentAmount != 0 && account.balance < intentAmount) {
       emit SuperDestinationExecutorReceivedButNotEnoughBalance(account);
       return;
   }
} else {
   IERC20 token = IERC20(tokenSent);
   if (intentAmount != 0 && token.balanceOf(account) < intentAmount) {
       emit SuperDestinationExecutorReceivedButNotEnoughBalance(account);
       return;
   }
}
```

However, this check fails to confirm that the account received new funds as a result of the current bridging event. If the **account already held sufficient funds** - either due to **prior activity**, **unrelated transfers**, or **replayed messages** - the execution would proceed **without validating the actual bridge inflow**.

This breaks the link between the user's off-chain intent and on-chain fund delivery. An attacker or buggy bridge adapter could exploit this by skipping or under-delivering the transfer while **reusing the account's existing balance to fulfill the check**. Since fund provenance is not enforced, **cross-bridge execution becomes unreliable.**

Impact:

- Users may unintentionally authorize execution paths backed by *old or unrelated balances*.
  Malicious or misconfigured adapters can trigger operations without delivering funds.
- In a worst-case scenario, bridged funds could be rerouted or withheld entirely, and the protocol would still consider the user's execution valid - causing financial loss or execution divergence.
- Signature binding does not mitigate this, as the `intentAmount` is not linked to a per-execution delta in balance.

Reproduction Steps:

1. A user bridges 10 ETH to their account using a Debridge or Across adapter.
2. The user's account already contains 15 ETH from previous interactions.
3. The adapter fails to deliver the new ETH or delivers less than 10 ETH.
4. The final balance is still > 10 ETH, so the check passes.
5. Execution continues, falsely assuming the bridged funds were received.

Recommendation:

- Track fund delta explicitly, In the adapter, capture the balance *before* and *after* the transfer.
- Use bridging receipts or proof, Introduce a minimal receipt layer (e.g., hashed transfer receipt or adapter-tracked nonce) that can be validated inside `processBridgedExecution`. This ensures every call is cryptographically tied to a unique bridge transfer.

M-03: Cross-Intent Interference via Global Balance Check

Severity: Medium
Category: Concurrency & Intent Isolation

Description:
Execution is triggered purely by `account.balance ≥ intentAmount`, so fills for Intent B can accidentally satisfy Intent A (and vice versa) if they interleave. Two intents of the same size but different fill schedules will race on the global balance, leading to premature or orphaned executions.

Example Timeline:

- **Intent A:** 3×33.33 transfers
- **Intent B:** 2×50 transfers

1. A_1 (33.33) → balance=33.33 (no fire)
2. A_2 (33.33) → balance=66.67 (no fire)
3. B_1 (50) → balance=116.67 → **fires B** early
4. A_3 (33.33)→ balance=50 (no fire, not enough balance)
5. B_2 (50) → balance=100 → **(**B's root used, revert**)**

Impact:

- **Premature executions** of one intent
- **Orphaned intents** whose final fills never fire
- **Unpredictable UX** and potential fund misuse

Recommendation:
Introduce a per-intent identifier and track fills exclusively against it.

L-01: Missing Reentrancy Protection in the DeBridge Adapter

Severity: Low
Category: Reentrancy

Description:
Even if you protect the executor, the adapter's own methods (onEtherReceived and onERC20Received) remain susceptible to reentrancy at the adapter level because they perform **an external call** (account.call or safeTransfer) **before doing anything else**.

A rogue fallback could reenter the adapter, triggering unexpected behaviors or stale state.

Recommendation:

Add OpenZeppelin's ReentrancyGuard to the adapter:

```
contract DebridgeAdapter is IExternalCallExecutor, ReentrancyGuard {
   function onEtherReceived(...) external payable nonReentrant returns (...) { … }
   function onERC20Received(...) external nonReentrant returns (…) { … } }
}
```

This ensures that no fallback from the user's account can reenter any adapter method.
By preventing reentrancy at both the executor and the adapter, you close off this DoS/vector and ensure the Merkle-root–signature flow can't be subverted by a malicious fallback.

I-01: Non-Standard Access Control Pattern in DeBridge Adapter

Severity: Info
Category: Code Quality & Best Practices

Description:
In `DebridgeAdapter.sol`, access control is implemented using a private function call instead of the standard Solidity modifier pattern:

```
function _onlyExternalCallAdapter() private view {
    if (msg.sender != externalCallAdapter) revert ONLY_EXTERNAL_CALL_ADAPTER();
}
```

While functionally equivalent to using modifiers, this pattern deviates from established Solidity conventions, reducing code readability and potentially making access control patterns harder to identify during code reviews and audits.

Recommendation:

Refactor to use the standard modifier pattern:

```
modifier onlyExternalCallAdapter() {
    if (msg.sender != externalCallAdapter) revert ONLY_EXTERNAL_CALL_ADAPTER();
    _;
}
function onERC20Received(...) external onlyExternalCallAdapter returns (...) {
   // Function logic
}
```

Apply this pattern consistently across all adapter implementations and similar access-controlled functions in the codebase

I-02: Outdated “Across” References in DebridgeAdapter

Severity: Info
Category: Coding Standards & Documentation

Description:

Several comments and error messages in DebridgeAdapter.sol still refer to “Across” (e.g. “Requires this adapter contract to hold the funds temporarily from Across”), a leftover from the Across adapter.

Recommendation:

Update all references from “Across” to “Debridge” (including comments and any error text) to avoid confusion.

I-03: Redundant Message Decoding in DeBridge Adapter

Severity: Info
Category: Gas Optimization & Code Efficiency

Description:
In `DebridgeAdapter.sol`, the incoming message payload is decoded twice - once partially in the entry point functions and then again completely in the message handling function:

```
// First partial decode in onEtherReceived
(,,, address account,) = _decodeMessage(_payload);
// ...transfer logic...
// Second complete decode in _handleMessageReceived
_handleMessageReceived(address(0), _payload);
// Inside _handleMessageReceived
function _handleMessageReceived(address tokenSent, bytes memory message) private {
   // Complete decode again
   (
       bytes memory initData,
       bytes memory executorCalldata,
       bytes memory sigData,
       address account,
       uint256 intentAmount
   ) = _decodeMessage(message);
   // Use decoded values
}
```

This double decoding pattern is inefficient as it performs the same operation twice on the same data, unnecessarily increasing gas costs and code complexity.

Recommendation:

- Consolidate the decoding operation to happen only once and pass the fully decoded data between functions:
- Restructure the `_handleMessageReceived` function to accept the decoded components directly instead of the raw message

6. Final Recommendations

Summary

1. **Prevent early Merkle-root poisoning:** Move the “mark root used” write below the balance check and keep the `return` on under-fills so partial transfers never consume the intent.
2. **Enforce explicit fund-delta checks:** In each adapter, snapshot pre- and post-transfer balances and require `delta ≥ intentAmount` to tie execution strictly to delivered funds.
3. **Isolate per-intent fills:** Assign each intent a unique ID and track its own received amount so Intent A and Intent B never race on the same global balance.
4. **Harden adapters with ReentrancyGuard:** Add OpenZeppelin’s `nonReentrant` to `onEtherReceived` to prevent fallback reentrancy at the adapter level.

Post-Remediation Testing

We recommend thoroughly testing the updated code to confirm changes work as intended and introduce no regressions:

- **Merit-Delta Verification:** Fuzz with pre-funded accounts, under-deliveries, exact fills, and over-deliveries to confirm only valid delta checks pass.
- **Adapter Reentrancy:** Deploy a malicious fallback account and verify that neither `onEtherReceived` nor `onERC20Received` can be reentered.
- **Intent Isolation:** Simulate parallel intents with overlapping and interleaved fills to ensure each intent fires exactly once.
- **Access Control Enforcement:** Attempt adapter calls from unauthorized addresses and confirm all entry points revert under the new modifier.
- **Decoding Integrity:** Test the single-decode refactor to ensure `(initData, executorCalldata, sigData, account, intentAmount)` is still extracted correctly and no valid payloads are rejected.

7. Conclusion

By implementing these focused recommendations, you’ll close the remaining medium- and low-severity gaps without altering core business logic. A brief post-remediation review is advised to validate that all tests pass and no new issues have been introduced.

8. Post-Fix Summary

Overview of Changes

- **M1:** https://github.com/superform-xyz/v2-contracts/pull/403
- **I1:** https://github.com/superform-xyz/v2-contracts/pull/406
- **I2:** https://github.com/superform-xyz/v2-contracts/pull/407
- **I3:** https://github.com/superform-xyz/v2-contracts/pull/408

- **Deferred Merkle-root Consumption (M-01):** Root now marks "used" only after reaching the intent amount.
- **Standardized Access Control (I-01):** Replaced private `_onlyExternalCallAdapter()` with a public `onlyExternalCallAdapter` modifier.
- **Single-Pass Decoding (I-02):** Bridge payloads are now decoded exactly once per adapter call, eliminating redundant `abi.decode`.
- **Corrected Protocol References (I-03):** All "Across" mentions in `DebridgeAdapter` updated to "Debridge."

Accepted Design Decisions

- **M2 and M3:** acknowledged (solvers can't send less; Also, if an account is pre-funded we are acknowledging that as a desired flow)
- **L1:** we tend to ACK this as well because the caller of that method can only be the externalCallAdapter. Adding the nonReentrant would add more gas consumption and not sure if this is really needed. However there's a PR for this here: https://github.com/superform-xyz/v2-contracts/pull/405

Remaining Security Concerns

- **Fund Provenance:** Reliance on final balances means provenance still isn't cryptographically tied to each transfer. Consider an optional delta-tracking mode for users needing strict bridged-fund guarantees.
- **Gas Overhead:** Introducing `nonReentrant` guards increases per-call gas - monitor its impact under heavy bridge usage.
- **Intent Serialization:** Current Merkle-root uniqueness plus off-chain bundler restriction suffices, but on-chain sequencing (nonces) could further guard against parallel-fill races in future iterations.

The above fixes greatly improve intent-flow safety, adapter robustness, and code clarity. With these fixes merged, Superform's cross-chain intent flow remains both powerful and performant - ready to scale to even the most demanding multi-chain yield strategies.