QuillAudits

AUDIT REPORT

November 2025

For

haha wallet

Table of Content

Executive Summary

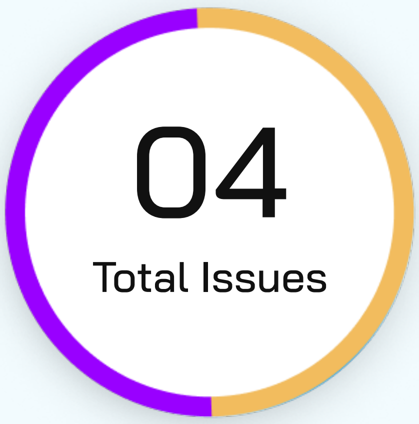
Project Name	Haha Wallet
Protocol Type	Wallet
Project URL	https://www.haha.me/
Overview	The protocol provides a modular smart-account system that lets users execute batched, signature-authorized transactions through a lightweight account contract, with strict nonce replay protection and enforced ERC-20 fee payments to a FeeManager. It is designed to attach stateless logic to EOAs (e.g., via EIP-7702), so execution rules, signature verification, and fee validation are all enforced without keeping per-account storage in the implementation. Alongside this, an upgradeable Merkle Distributor handles secure token/ETH claims using Merkle proofs, ensuring only eligible users can withdraw assigned amounts while preventing double-claims. Together, the components enable secure meta-transactions, deterministic fee handling, and efficient token distribution.
Audit Scope	The scope of this Audit was to analyze the Haha Wallet Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Permutize/smart-accounts https://github.com/Permutize/merkletree-smart-contract
Branch	main
Contracts in Scope	BaseAccount.sol, MetaAccount.sol, FeeManager.sol, IncrementalNonces.sol, CallHash.sol, MerkleTokenDistributor.sol, MerkleTokenDistributorUpgradeable.sol
Commit Hash	470508622ee6d7c4971fc67c0601fecbc9dbc4c4, 8a1bf9cc9db45279261402931c9cd0efb93850cc
Language	Solidity
Blockchain	Monad, Ethereum, BSC, Polygon, Base, Arbitrum, Avalanche

Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	18th November 2025
Updated Code Received	27th November 2025
Review 2	27th November 2025
Fixed In	7d27e091c396e0213af10fcd413677cdf5fd0acd 5bc1759b0077302c3befe0f5537fe4f7810cb41f

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0(0.0%)
🟥	High	0(0.0%)
🟧	Medium	0(0.0%)
🟨	Low	2 (50.0%)
🟪	Informational	2 (50.0%)

Severity

Issues	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	1	1
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	1	1

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Consider using Ownable2Step instead	Low	Resolved
2	Contract is not compatible with Fee on Transfer Tokens	Low	Acknowledged
3	Consider adding timelock while changing MerkleRoot by admin	Informational	Resolved
4	Expected fee call format (ERC20.transfer to feeManager with amount) not documented in NatSpec	Informational	Acknowledged

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw
- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open Security vulnerabilities identified that must be resolved and are currently unresolved.	**Resolved** These are the issues identified in the initial audit and have been successfully fixed.
Acknowledged Vulnerabilities which have been acknowledged but are yet to be resolved.	**Partially Resolved** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	🟥 High	🟧 Medium	🟨 Low
🟥 High	Critical	High	Medium
🟧 Medium	High	Medium	Low
🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Low Severity Issues

Consider using Ownable2Step instead

Resolved

Path

MerkleDistributor.sol, MerkleDistributorUpgradeable.sol, BaseAccount.sol, IncrementalNonces.sol, FeeManager.sol

Description

The Ownable2Step prevents the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner's permissions actively accept via a contract call of its own.

Recommendation

Consider using Ownable2Step from OpenZeppelin Contracts to enhance the security of your contract ownership management. This contract prevents the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call.

Contract is not compatible with Fee on Transfer Tokens

Acknowledged

Path

MerkleDistributor.sol, MerkleDistributorUpgradeable.sol, BaseAccount.sol

Description

_isValidFeeCall checks that the first call in the batch is an ERC-20 transfer to the FeeManager with an amount between minFeeCost and maxFeeCost. However, the contract assumes that the token will transfer exactly the specified amount, which is not true for fee-on-transfer tokens.

FOT tokens deduct a percentage of the transfer amount as a burn or fee, meaning: The actual amount received by the FeeManager is not equal to the amount sent. _isValidFeeCall verification passes even though the FeeManager receives less than expected. This results in underpayment or bypass of required fees.

Recommendation

Explicitly disallow fee-on-transfer tokens by enforcing a strict ERC-20 compliance check or by maintaining an allowlist of supported tokens.

Haha Wallet Team's Comment

We will not support FOT tokens.

Informational Issues

Consider adding timelock while changing MerkleRoot by admin

Resolved

Description

Currently setMerkleRoot(bytes32) is an owner-only function that can be executed immediately. There is no on-chain delay or timelock guarding the update. An owner compromise or accidental/rogue update could change the distribution rules instantly, causing tokens to be redirected, claims invalidated, or users to lose access without notice. Immediate root changes also reduce transparency and stop off-chain actors from reacting (e.g., canceling queued claims).

Recommendation

Add a timelock (e.g., 24–72 hours) for Merkle root updates: when setMerkleRoot is called propose the new root and record a timestamp + delay; only allow finalization after the delay.

Expected fee call format (ERC20.transfer to feeManager with amount) not documented in NatSpec

Acknowledged

Description

The contract assumes fee payments come as an ERC20.transfer call to the FeeManager with a specified amount, but this expected calldata format is not described in the contract NatSpec or public docs.

Recommendation

Add a concise NatSpec comment (and an integration note) specifying the exact expected fee-call ABI, e.g.:

Note to Users/Trust Assumptions

Centralization Risk

The protocol carries a degree of centralization risk because key operational controls—such as updating the Merkle root, configuring supported fee tokens, and managing fee parameters—are fully governed by the designated owner or admin. If this privileged role is compromised, misused, or operated incorrectly, the admin can alter distributions, disable or manipulate fee-token settings, or withdraw assets held in the distributor. Since users rely on the integrity of these configurations for correct execution and fair distribution, the trust model assumes that the admin is honest, secure, and properly governed (e.g., via multisig or timelock). Strengthening these controls is recommended to reduce single-party risk.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Rejects batch when first call is not a valid ERC20 fee transfer
- ✓ Accepts batch only when first call is a valid fee payment (correct selector, recipient = FeeManager, token enabled, amount in range)
- ✓ Executes batch with a valid signature
- ✓ Reverts entire batch if any call reverts
- ✓ simulateBatch must revert when executed on-chain (tx.origin != 0)
- ✓ Allows claim with valid proof and marks index as claimed
- ✓ Only owner can update merkle root

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
BaseAccount.sol	execute(Batch batch, bytes signature)	**Inputs** • batch • Control: caller / relayer (attacker can craft) • Constraints: deadline must be > block.timestamp; calls.length > 0 enforced by modifier • Impact: The calls executed on behalf of the account (can transfer funds, change state) • signature • Control: user (signer) or attacker providing garbage • Constraints: must match EIP-712 hashed batch and owner key • Impact: Authorization; if valid allows execution

Contract	Function	Threats
		Intended branches / code coverage (tests) • Valid signature + valid nonce → calls executed successfully (all succeed) • Valid signature but one internal call reverts → whole batch reverts (atomic) • Invalid signature → revert InvalidSignature() before nonce consumption (ensure ordering) • Expired deadline → revert InvalidDeadline() • Replay attempts (reuse nonce) → revert via NONCE_MANAGER.useCheckedNonce **Negative tests** • Submit invalid signature (reverts, nonce unchanged) • Submit with expired deadline (reverts) • Submit empty calls (revert by modifier) • Replay same batch with same nonce (should revert)
	simulateBatch(Batch batch, bytes signature)	Purpose: Allow relayers/ wallets to preview execution (revert reasons, estimate gas) without changing state. **Inputs** • batch, signature — same as execute, but intended for eth_call.

Contract	Function	Threats
		Intended branches / code coverage (tests) • eth_call simulation returns revert reason for failing internal call (no state changes) • Simulation must revert if called on-chain (tx.origin != address(0) check) • Simulation can accept invalid signature / nonce and still run for preview (verify that simulation won't write) **Negative tests** • Calling simulate on-chain must revert (safety) • Simulation with malformed calldata should return the same revert reason as execute eth_call would
	withdrawToken(address token, address to, uint256 amount)	Purpose: Owner-only withdrawal of ETH or ERC-20 tokens from the account. **Inputs** • token (0 for ETH), to, amount — owner-controlled inputs **Intended branches / code coverage (tests)** • Owner withdraws ETH successfully Owner withdraws ERC-20 via • SafeERC20 successfully • Non-owner call reverts

Contract	Function	Threats
		Negative tests • Non-owner cannot withdraw • Withdraw amount > balance should revert/handle gracefully
MetaAccount.sol	_isValidFeeCall(Call call)	Purpose: Ensure the first call in a batch is a valid ERC-20 transfer to the configured FeeManager with amount inside allowed bounds. **Inputs** • call.to (token), call.data (calldata): selector + args • Control: user/relayer constructing batch • Constraints: token must be listed/enabled in FeeManager; calldata must be ERC-20 • transfer(recipient, amount); recipient must be FeeManager; amount within min/max fee cost • Impact: Decides whether relayer can be paid; prevents fee bypass

Contract	Function	Threats
		Intended branches / code coverage (tests) • Valid ERC-20 transfer(in FeeManager, amount within range) → returns true • Wrong selector → false • Recipient != FeeManager → false • Amount < minFeeCost or > maxFeeCost → false • Disabled token in FeeManager → false **Negative tests** • Malformed calldata lengths (too short) should reject gracefully • Nonstandard ERC-20 (no bool return / fee-on-transfer) should be handled/flagged
MerkleDistributorUpgradeable	claim(index, account, token, amount, proof)	Purpose: Allow eligible user to claim preallocated amount via Merkle proof and mark index claimed. **Inputs** • index, account, token, amount, proof • Control: claimant (user) supplies data and proof • Constraints: proof must validate against merkleRoot; index must be unclaimed • Impact: transfers funds to account and marks index claimed

Contract	Function	Threats
		Intended branches / code coverage (tests) • Valid proof, unclaimed index → mark claimed + transfer funds (ETH or ERC-20) • Claiming same index twice → revert Already claimed • Invalid proof → revert Invalid proof **Negative tests** • Claim with invalid proof (revert) • Try claim twice (revert) • Claim with manipulated amount or account not matching leaf (revert)
	claimMany	Purpose: Allow batch claiming of multiple indices in one transaction. **Inputs** • arrays indices, tokens, amounts, proofs and account **Intended branches / code coverage (tests)** • Valid batch where all proofs valid → all marked claimed + transfers succeed • If any claim in batch fails, entire tx reverts (atomicity) — ensure correct behavior

Contract	Function	Threats
		Negative tests • Batch with mismatched array lengths → revert Length mismatch • Batch too large → OOG (test for large batch behavior) • One invalid proof in batch leads to entire batch revert

Closing Summary

In this report, we have considered the security of Haha Wallet. We performed our audit according to the procedure described above.

No critical issues in Haha Wallet, just 2 issues of Low and 2 Informational severity were found. The team acknowledged two and resolved the remaining issues.

Disclaimer

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

November 2025

For

haha wallet

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com