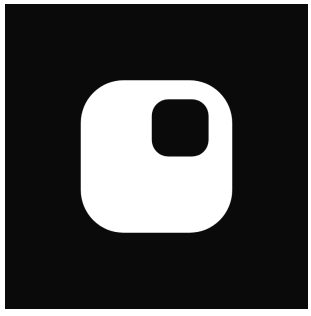
KANN AUDITS

SECURITY REVIEW

FOR GLOAM



PREPARED FOR
Gloam

PREPARED BY
Kann Audits

DATE REPORTED
1 November 2026

INSIDE THIS REPORT

CONTENTS

01	AUDIT SUMMARY	03
02	PROTOCOL OVERVIEW	04
03	SCOPE	05
04	AUDITORS	06
05	METHODOLOGY	07
06	RISK CLASSIFICATION	08
07	FINDINGS SUMMARY	09
08	DETAILED FINDINGS	10
L-01	Global Deposit Cap Exhaustion Can Deny Deposits and Disrupt Swap Refunds	10
L-02	Withdrawal Proofs Do Not Bind the Applicable Fee Rate or Minimum Received Amount	13
L-03	Merkle Tree Capacity Exhaustion Permanently Disables New Note Creation	16
I-01	Disabling Withdrawal Fees Prevents Collection of Previously Accrued Fees	18
I-02	Governance-Controlled Verifier Replacement Can Compromise Proof Validation	20
I-03	Withdrawals to the Pool Address Can Permanently Lock User Funds	22
I-04	Timelock Allows Scheduling Shield Verifier Updates That Cannot Execute	24
I-05	Expensive Zero-Knowledge Proof Verification Precedes Basic Deposit Validation	26
I-06	Merkle Root History Storage Introduces Additional Gas Overhead	28
I-07	Missing Constructor Validation Allows Invalid Proof Verifier Dependencies	30
I-08	ERC-20 tokens sent directly to the pool cannot be recovered	32
09	DISCLAIMER	33
10	ABOUT KANN AUDITS	34

01 - OVERVIEW

AUDIT SUMMARY

Gloam engaged Kann Audits to perform a security review of the Gloam smart contracts. From 26 – 30 October 2026, a team of 5 security researchers performed a manual review of the in-scope contracts and additionally triaged findings generated by the Kann AI Security Model. All findings identified during the assessment are documented in this report.

TOTAL FINDINGS

11 ISSUES

CRITICAL	–
HIGH	–
MEDIUM	–
LOW	03
INFORMATIONAL	08

ENGAGEMENT DETAILS

PREPARED FOR	Gloam
PREPARED BY	Kann Audits
DATE REPORTED	1 November 2026
LANGUAGE	Solidity
CHAIN	Robinhood Chain, Tempo, Base, Ethereum
REVIEW PERIOD	26 – 30 October 2026
SCOPE	668 nSLOC
METHODS	Manual review, Kann AI Security Model

02 - WHAT THE PROTOCOL DOES

PROTOCOL OVERVIEW

Gloam is a private payment protocol for stablecoins and other supported ERC-20 assets on Robinhood Chain, Tempo, Base and Ethereum. The Base and Ethereum deployments use the same vault bytecode and also support native ETH. Users deposit assets into a shared vault, hold balances as private notes and authorize private transfers or withdrawals with zero-knowledge proofs.

Gloam supports individual payments, team payroll and payments by AI agents. Users can selectively prove facts about their balances or payments without revealing their full financial activity.

03 - WHAT WE REVIEWED

SCOPE

REPOSITORY	github.com/cryptoduke01/gloam
AUDITED COMMIT	aac9e185bbeaa6b74311236c1cc57077d928f407

#	FILE	NSLOC
01	contracts/src/ShieldPoolCore.sol	364
02	contracts/src/ShieldPoolPayments.sol	15
03	contracts/src/lib/ChangeTimelock.sol	75
04	contracts/src/lib/IncrementalMerkleTreePoseidon.sol	97
05	contracts/src/lib/NoteLibPoseidon.sol	26
06	contracts/src/verifiers/DualProofVerifier.sol	22
07	contracts/src/interfaces/IAggregatorV3.sol	14
08	contracts/src/interfaces/IShieldPool.sol	41
09	contracts/src/interfaces/IVerifier.sol	7
10	contracts/src/lib/IPoseidon.sol	7
	Total	**668**

nSLOC excludes blank and comment-only lines in the ten listed files at the audited commit.

Scope boundaries: optional files GloamVaultSwapV4.sol, GloamVaultSwapTempo.sol and GloamPayMemo.sol are not included in the primary scope. The v2 router (GloamVaultSwapV4b.sol, GloamVaultSwap.sol and SwapEscrow.sol), generated final verifiers, legacy pools, tests and vendored dependencies are excluded. L-01 cites router integration behavior without extending full audit coverage to that router.

04 - THE TEAM ON THIS ENGAGEMENT

AUDITORS

KA	Kann	SECURITY RESEARCHER
AL	AlexScherbatyuk	SECURITY RESEARCHER
DG	DGrayArea	SECURITY RESEARCHER
NY	NYXARIS	SECURITY RESEARCHER
SA	SarveshLimaye	SECURITY RESEARCHER

05 - HOW WE WORK

METHODOLOGY

An engagement with Kann Audits involves a team of security researchers performing independent reviews of the codebase at the start of the engagement, followed by collaborative review sessions where auditors share findings and explore additional attack vectors to improve overall coverage.

The review included a line-by-line manual inspection of the in-scope codebase, along with analysis of the protocol architecture, access control mechanisms and state transitions. The team also assessed the code against common smart contract security best practices and industry standards.

STEP BY STEP

OUR SECURITY PROCESS

01 **Kickoff & Context**
A kickoff call to understand architecture, trust boundaries, threat assumptions and intended behaviour.

02 **Scope & Preparation**
Confirm the commit, files in scope, dependencies and the riskiest areas. We open a private channel between your team and our auditors.

03 **AI First Scan**
Kann AI, our proprietary model built to find security bugs, scans the codebase before the manual review starts.

04 **Expert Security Review**
Line-by-line analysis of exploit paths, invariant breaks, edge cases, accounting and integration risks.

05 **Findings & Proofs**
Each issue is confirmed, rated and backed by a proof of concept or a precise reproduction path.

06 **Mitigation Review**
We review every fix, confirm it resolves the root cause and flag partial or incorrect mitigations.

COVERAGE

THE REVIEW SPECIFICALLY CONSIDERED

- Testing against common and uncommon attack vectors
- Contract logic matching the client's intended behaviour
- Manual line-by-line review by experienced auditors
- Architecture and design analysis for secure component interactions and clear trust boundaries
- Compliance with industry best practices and known smart contract security standards
- Cross-referencing patterns with industry-standard implementations
- Access control verification
- State transitions and logic, to detect inconsistencies, unexpected behaviour or exploitation paths

06 - HOW WE RATE ISSUES

RISK CLASSIFICATION

Severity combines the **impact** of an issue if exploited with the **likelihood** that the conditions for exploitation are met.

CRITICAL	Directly exploitable with few or no preconditions, leading to theft or permanent loss of funds or loss of control of the system. Must be fixed before deployment.
HIGH	Directly exploitable vulnerabilities that can lead to significant material loss or critical impact on the system and require immediate fixing.
MEDIUM	Moderate loss of funds or impact on multiple users, but only under specific conditions. Exploitable without privileged access, but typically requires certain assumptions or edge cases.
LOW	Issues with limited impact that do not put funds directly at risk, such as edge cases or deviations from documented behaviour.
INFORMATIONAL	Non-exploitable findings such as best-practice deviations, cosmetic issues or documentation and compliance notes. They do not need urgent remediation but improve code quality and maintainability.

07 - ALL ISSUES

FINDINGS SUMMARY

CRITICAL	HIGH	MEDIUM	LOW	INFO
-	-	-	03	08

TOTAL 11 FIXED 02 ACKNOWLEDGED 09

Fixed indicates planned remediation, not live fixes. L-01: next router; L-02: next vault.

ID	TITLE	SEVERITY	STATUS
L-01	Global Deposit Cap Exhaustion Can Deny Deposits and Disrupt Swap Refunds	LOW	FIXED
L-02	Withdrawal Proofs Do Not Bind the Applicable Fee Rate or Minimum Received Amount	LOW	FIXED
L-03	Merkle Tree Capacity Exhaustion Permanently Disables New Note Creation	LOW	ACKNOWLEDGED
I-01	Disabling Withdrawal Fees Prevents Collection of Previously Accrued Fees	INFORMATIONAL	ACKNOWLEDGED
I-02	Governance-Controlled Verifier Replacement Can Compromise Proof Validation	INFORMATIONAL	ACKNOWLEDGED
I-03	Withdrawals to the Pool Address Can Permanently Lock User Funds	INFORMATIONAL	ACKNOWLEDGED
I-04	Timelock Allows Scheduling Shield Verifier Updates That Cannot Execute	INFORMATIONAL	ACKNOWLEDGED
I-05	Expensive Zero-Knowledge Proof Verification Precedes Basic Deposit Validation	INFORMATIONAL	ACKNOWLEDGED
I-06	Merkle Root History Storage Introduces Additional Gas Overhead	INFORMATIONAL	ACKNOWLEDGED
I-07	Missing Constructor Validation Allows Invalid Proof Verifier Dependencies	INFORMATIONAL	ACKNOWLEDGED
I-08	ERC-20 tokens sent directly to the pool cannot be recovered	INFORMATIONAL	ACKNOWLEDGED

08 - ISSUE BY ISSUE

DETAILED FINDINGS

This report documents 11 findings. Superseded issue versions are excluded. Project responses accompany each finding. Acknowledged items include planned remediation and existing operational safeguards. For the two findings labeled Fixed, neither fix is live: L-01 is planned for the next router deployment, and L-02 for the next vault deployment.

L-01 Global Deposit Cap Exhaustion Can Deny Deposits and Disrupt Swap Refunds

SEVERITY LOW STATUS FIXED

LOCATION

ShieldPoolCore.sol - _shield(); GloamVaultSwap.sol - swap(), _deposit().

DESCRIPTION

The protocol enforces deposit limits through a global outstanding-balance cap for each supported asset.

When depositLimitsOn is enabled, every new deposit is checked against the configured per-deposit and total deposit limits.

However, the total limit applies collectively to all users without individual allocations or reserved capacity for protocol-critical operations.

Consequently, a user with sufficient capital can fill the entire available deposit capacity through multiple legitimate deposits.

Once the global cap is reached, subsequent deposits of that asset revert until existing deposits are withdrawn or the configured limit is increased.

VULNERABLE CODE

```
if (depositLimitsOn) {
    DepositLimit memory lim = depositLimit[asset];

    if (
        amount > lim.perDeposit ||
        deposited[asset] + amount > lim.total
    ) {
        revert OverDepositLimit();
    }
}
```

This limitation also affects the swap router, which deposits swap output and refund notes using the same shieldBound() mechanism.

```
function _deposit(Deposit calldata d) private {
    _approve(d.asset, address(vault), d.amount);

    vault.shieldBound(
        d.asset,
        d.amount,
        d.commitment,
        d.proof
    );

    _approve(d.asset, address(vault), 0);
}
```

The refund branch depends on this function:

```
if (refunding) {
    if (available < r.amount)
        revert BadRefund();

    _deposit(r);
    _trySweep(s.asset);

    emit VaultSwapRefund(
        s.nullifier,
        r.commitment,
        r.asset,
        r.amount
    );

    return 0;
}
```

Therefore, the deposit cap applies not only to ordinary user deposits but also to the creation of shielded refund notes.

If a swap's withdrawal is executed separately and its funds enter escrow, an attacker who subsequently occupies the freed deposit capacity can potentially prevent the refund from being deposited back into the vault.

EXAMPLE SCENARIO

The protocol has a total USDC deposit cap of 25,000 USDC.

An attacker fills the available capacity using legitimate deposits.

Other users attempting to deposit USDC encounter OverDepositLimit().

A swap refund that requires a new USDC commitment also encounters the same restriction.

If the original withdrawal has already been completed separately, the escrowed funds cannot be returned through that refund transaction until capacity becomes available.

The final scenario requires an integration test demonstrating that an external attacker can reliably obtain and exploit the required transaction ordering.

IMPACT

An attacker can deny other users the ability to deposit an affected asset by monopolizing its global deposit capacity.

The same restriction can prevent swap output deposits and may temporarily block recovery of funds held in swap escrow.

Because raising deposit limits requires a three-day governance delay after setup, the protocol cannot immediately increase capacity to resolve the disruption.

RECOMMENDATION

Separate ordinary deposit capacity from the capacity required for internal settlement and refund operations.

Ensure that swap refunds can be completed regardless of the current global deposit utilization, without allowing ordinary deposits to bypass the protocol's intended risk limits.

Consider implementing safeguards against excessive capacity concentration and monitoring deposit utilization to detect potential service disruption.

REMEDIATION

FIXED

GLOAM TEAM

Status: Confirmed, Low. Higher limits remove it for listed assets. The refund path gets fixed at the next router deploy.

Planned remediation, not live: The refund-path fix is planned for the next router deployment.

L-02 Withdrawal Proofs Do Not Bind the Applicable Fee Rate or Minimum Received Amount

SEVERITY LOW STATUS FIXED

LOCATION

ShieldPoolCore.sol - _requireUnshieldProof(), unshield(), _withdrawFee().

DESCRIPTION

The protocol's withdrawal proof verifies the validity of a private note and binds the withdrawal to a specific Merkle root, nullifier, asset, recipient, and amount.

However, the proof does not commit to the applicable withdrawal fee or the minimum amount the recipient must receive.

Instead, the contract calculates the withdrawal fee using the currently configured withdrawFeeBps at transaction execution.

Consequently, a withdrawal proof generated under one fee configuration remains valid if the fee rate subsequently changes.

This can cause the amount ultimately received by the user to differ from the amount expected when the proof was generated.

RELEVANT CODE

```
uint256[] memory inputs = new uint256[](5);

inputs[0] = uint256(root);
inputs[1] = uint256(nullifier);
inputs[2] = uint256(uint160(asset));
inputs[3] = amount;
inputs[4] = uint256(uint160(to));

if (!verifier.verify(proof, inputs))
    revert InvalidProof();
```

The fee is calculated independently after verification:

```
uint256 fee = _withdrawFee(amount);

deposited[asset] -= amount;

if (fee ≠ 0)
    feesOwed[asset] += fee;

_pushAsset(asset, to, amount - fee);
```

The applicable fee depends on mutable contract state:

```
function _withdrawFee(
    uint256 amount
) internal view returns (uint256) {
    uint256 bps = withdrawFeeBps;

    if (bps == 0 || feeRecipient == address(0))
        return 0;

    return
        (amount / BPS) * bps +
        ((amount % BPS) * bps) / BPS;
}
```

EXAMPLE SCENARIO

A user generates a proof for a 1,000 USDC withdrawal when the configured fee rate is 0.25%.

The user expects to receive 997.50 USDC.

Before the withdrawal is executed, the governance-controlled fee rate increases to 1%.

The same proof remains valid because the fee rate is not one of its public inputs.

Upon execution, the contract deducts 10 USDC instead of 2.50 USDC, resulting in the recipient receiving 990 USDC.

The rate increase remains subject to the protocol's governance restrictions, and withdrawal fees are capped at 1%.

IMPACT

Users may receive fewer assets than expected if the withdrawal fee changes between proof generation and transaction execution.

This introduces a fee-slippage risk, particularly for delayed transactions or withdrawals submitted through relayers.

The maximum discrepancy is bounded by the protocol's configured withdrawal fee cap.

RECOMMENDATION

Introduce an execution-time protection allowing users to specify the maximum acceptable withdrawal fee or minimum amount they are willing to receive.

Ensure that withdrawals exceeding the user's accepted fee tolerance revert rather than executing with an unexpected net amount.

Consider exposing the current fee configuration clearly to wallets and relayers before proof generation and transaction submission.

REMEDIATION FIXED

GLOAM TEAM

Status: Fixed. The fee is capped at 1% and any raise sits in a public 3-day timelock, so the gap is bounded and visible in advance. We'll bind the fee or a minimum received amount in the next vault's proof.

Planned remediation, not live: Binding the withdrawal fee or a minimum received amount in the proof is planned for the next vault deployment.

L-03 Merkle Tree Capacity Exhaustion Permanently Disables New Note Creation

SEVERITY LOW STATUS ACKNOWLEDGED

LOCATION

IncrementalMerkleTreePoseidon.sol - insert(); ShieldPoolCore.sol - transfer().

DESCRIPTION

The protocol uses a fixed-depth Poseidon Merkle tree with a maximum capacity of 1,048,576 leaves.

Each deposit inserts one new commitment, while each private transfer inserts two commitments.

Previously spent notes remain in the tree, and their associated leaf positions are never reclaimed.

Because private transfers can repeatedly consume new leaf positions without requiring additional deposits of equivalent value, a user can progressively exhaust the available tree capacity by performing valid private transfers.

Once the maximum capacity is reached, subsequent attempts to insert commitments revert with TreeFull().

VULNERABLE CODE

```
uint256 internal constant DEPTH = 20;

uint256 internal constant MAX_LEAVES =
    uint256(1) << DEPTH;
```

The insertion function enforces the fixed maximum:

```
function insert(
    Tree storage self,
    uint256 leaf
) internal returns (uint256 index) {
    if (!self.initialized)
        revert NotInitialized();

    if (leaf == 0)
        revert ZeroLeaf();

    if (self.nextIndex >= MAX_LEAVES)
        revert TreeFull();

    index = self.nextIndex;

    // Merkle tree update logic

    self.nextIndex = index + 1;
}
```

Meanwhile, every private transfer consumes two positions:

```
for (uint256 i = 0; i < 2; i++) {
    if (newCommitments[i] == bytes32(0))
        revert ZeroCommitment();

    if (commitmentSeen[newCommitments[i]])
        revert DuplicateCommitment();

    commitmentSeen[newCommitments[i]] = true;

    tree.insert(uint256(newCommitments[i]));
}
```

Approximately 524,288 private transfers would be required to fill an otherwise empty tree when each transfer consumes two leaves.

IMPACT

Once the Merkle tree reaches its maximum capacity, new deposits and private transfers become unavailable.

Other operations that depend on inserting new commitments, including certain swap operations, may also fail.

Existing users can still withdraw through unshield() because withdrawals do not insert new leaves.

Although deliberate exhaustion involves substantial transaction costs, the same limitation can eventually occur through legitimate protocol usage.

RECOMMENDATION

Introduce a safe mechanism for transitioning to new Merkle trees or pool instances before capacity is exhausted.

Preserve the ability to spend existing notes during migration and ensure that the transition does not interrupt deposits or private transfers.

Monitor tree utilization and establish operational thresholds for initiating migrations before reaching maximum capacity.

REMEDIATION ● ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. By design. Robinhood Chain is at about 550 of 1,048,576 leaves. We'll monitor it and move to a new vault well before it fills.

I-01 Disabling Withdrawal Fees Prevents Collection of Previously Accrued Fees

SEVERITY INFORMATIONAL STATUS ACKNOWLEDGED

LOCATION

ShieldPoolCore.sol - _setWithdrawFee(), collectFees().

DESCRIPTION

The protocol allows the owner to disable withdrawal fees by setting the fee rate and recipient address to zero.

However, the configuration process does not account for fees accumulated before the change.

The _setWithdrawFee() function allows the recipient address to be cleared whenever the new withdrawal fee rate is zero, regardless of whether outstanding balances remain in feesOwed.

Meanwhile, collectFees() requires a nonzero feeRecipient before processing any fee withdrawal.

As a result, disabling the fee mechanism can make previously accrued fees temporarily inaccessible.

VULNERABLE CODE

```
function _setWithdrawFee(
    uint16 bps,
    address recipient
) internal {
    if (bps > MAX_WITHDRAW_FEE_BPS)
        revert FeeTooHigh();

    if (bps != 0 && recipient == address(0))
        revert ZeroAddress();

    withdrawFeeBps = bps;
    feeRecipient = recipient;

    emit WithdrawFeeSet(bps, recipient);
}
```

The collection function subsequently rejects the configuration:

```
function collectFees(
    address asset
) external nonReentrant {
    address recipient = feeRecipient;

    if (recipient == address(0))
        revert ZeroAddress();

    uint256 amount = feesOwed[asset];

    if (amount == 0)
        revert NothingToCollect();

    feesOwed[asset] = 0;
    _pushAsset(asset, recipient, amount);
}
```

EXAMPLE SCENARIO

The protocol accumulates 500 USDC in withdrawal fees.

The owner disables fees by setting the rate and recipient to zero.

The outstanding 500 USDC remains recorded in feesOwed.

Subsequent calls to collectFees() revert with ZeroAddress().

The owner must restore a valid fee recipient before collecting the outstanding amount.

IMPACT

Previously accrued protocol revenue becomes temporarily inaccessible when the withdrawal fee recipient is cleared.

After the setup phase, restoring a valid recipient requires a new governance change subject to the three-day timelock, introducing an unnecessary delay in fee collection.

RECOMMENDATION

Ensure that disabling future withdrawal fees does not prevent collection of previously accrued fees.

Consider preserving the existing fee recipient when fees are disabled or requiring outstanding fees to be settled before allowing the recipient to be removed.

REMEDIATION ● ACKNOWLEDGED

GLOAM TEAM

Info. lowerWithdrawFee(0) keeps the recipient, so owed fees stay collectable. It only happens through setWithdrawFee(0, address(0)), and we won't use that.

Classified as Informational in line with the Gloam response.

I-02 Governance-Controlled Verifier Replacement Can Compromise Proof Validation

SEVERITY ● INFORMATIONAL STATUS ● ACKNOWLEDGED

LOCATION

ShieldPoolCore.sol - _setVerifier(), _requireUnshieldProof(). **Severity context:** Informational - Administrative Trust Risk.

DESCRIPTION

The protocol permits the owner to replace the main proof verifier through the governance timelock.

However, the verifier configuration process does not validate whether the new contract correctly implements the expected proof-verification logic or remains compatible with existing private notes.

Consequently, an incorrectly configured verifier may reject legitimate proofs and prevent users from accessing their funds until the configuration is corrected.

More importantly, the protocol relies on governance to select trustworthy verifier implementations.

If governance is compromised or intentionally installs a malicious verifier, the proof-validation mechanism may accept unauthorized withdrawal requests.

RELEVANT CODE

```
function _setVerifier(
    address verifier_
) internal {
    verifier = IVerifier(verifier_);

    emit VerifierSet(verifier_);
}
```

Withdrawal authorization relies on the configured verifier:

```
if (address(verifier) == address(0))
    revert VerifierNotSet();

if (!tree.isKnownRoot(uint256(root)))
    revert UnknownRoot();

uint256[] memory inputs = new uint256[](5);

inputs[0] = uint256(root);
inputs[1] = uint256(nullifier);
inputs[2] = uint256(uint160(asset));
inputs[3] = amount;
inputs[4] = uint256(uint160(to));

if (!verifier.verify(proof, inputs))
    revert InvalidProof();
```

The governance timelock provides a three-day notice period before changes can take effect but does not independently validate verifier correctness.

IMPACT

An incompatible verifier can temporarily disrupt legitimate withdrawals and private transfers.

A malicious verifier installed through compromised governance could undermine proof-based authorization and potentially permit unauthorized withdrawals of pooled funds.

The issue represents an important administrative trust assumption rather than a demonstrated permissionless exploit.

RECOMMENDATION

Establish a secure verifier-upgrade policy requiring thorough review and compatibility validation of replacement implementations.

Consider limiting upgrades to previously approved verifier contracts and requiring additional authorization for changes affecting withdrawal security.

Ensure that users receive advance notice of verifier changes and retain an opportunity to withdraw before potentially incompatible upgrades take effect.

REMEDIATION

● ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. Known. We've built an owner contract that blocks it, and handover waits on an outside review.

I-03 Withdrawals to the Pool Address Can Permanently Lock User Funds

SEVERITY ● INFORMATIONAL STATUS ● ACKNOWLEDGED

LOCATION

ShieldPoolCore.sol - unshield(), _pushAsset().

DESCRIPTION

The unshield() function prevents withdrawals to the zero address but does not validate whether the recipient is the pool contract itself.

When a user withdraws to the pool address, the contract verifies the withdrawal proof, marks the associated nullifier as spent, and decreases the recorded deposited balance. However, a standard ERC-20 transfer from the pool to itself leaves the contract's actual token balance unchanged.

As the corresponding note has already been spent, the transferred amount is no longer represented by a spendable note. The contract also lacks a general mechanism for recovering surplus ERC-20 balances.

VULNERABLE CODE

```
function unshield(
    bytes calldata proof,
    bytes32 root,
    bytes32 nullifier,
    address asset,
    address to,
    uint256 amount
) external override nonReentrant {
    if (to == address(0)) revert ZeroAddress();

    if (amount == 0) revert InvalidAmount();
    if (spent[nullifier]) revert AlreadySpent();

    _requireUnshieldProof(
        proof, root, nullifier, asset, to, amount
    );

    spent[nullifier] = true;

    if (deposited[asset] < amount)
        revert InsufficientPoolBalance();

    uint256 fee = _withdrawFee(amount);
    deposited[asset] -= amount;

    if (fee != 0) feesOwed[asset] += fee;

    _pushAsset(asset, to, amount - fee);
}
```

The _pushAsset() function does not distinguish between an external recipient and the pool itself.

```
(bool success, bytes memory data) = asset.call(
    abi.encodeWithSelector(
        IERC20.transfer.selector,
        to,
        amount
    )
);
```

IMPACT

Users can permanently lose access to their funds by specifying the pool contract as their withdrawal recipient.

Although the issue requires an incorrect withdrawal destination, the affected note becomes unspendable, and the remaining ERC-20 tokens cannot be recovered through the pool's normal operations.

RECOMMENDATION

Prevent withdrawals to the pool contract's own address by validating the recipient before processing the withdrawal proof.

Apply the restriction at the smart contract level to protect users regardless of the frontend, wallet, or SDK used to submit the transaction.

REMEDIATION ACKNOWLEDGED

GLOAM TEAM

Status: Confirmed, Informational. The SDK, app and relay already refuse this, and we'll add the contract check in the next vault.

I-04 Timelock Allows Scheduling Shield Verifier Updates That Cannot Execute

SEVERITY INFORMATIONAL STATUS ACKNOWLEDGED

LOCATION

ShieldPoolCore.sol - _changeArgWords(), _setShieldVerifier().

DESCRIPTION

The governance timelock explicitly permits scheduling changes to the shield verifier through the setShieldVerifier() selector.

However, the underlying verifier configuration function permanently prevents the verifier from being replaced once a nonzero address has been assigned.

Consequently, after initialization, the owner can schedule a verifier update, wait through the governance delay, and attempt to execute the change, but execution will always revert.

This creates an inconsistency between the operations permitted by the governance interface and the operations that can actually be executed.

RELEVANT CODE

```
function _changeArgWords(
    bytes4 selector
) internal pure virtual override returns (uint256) {
    if (
        selector ==
        ShieldPoolCore.setShieldVerifier.selector
    ) return 1;

    // Other selectors omitted
    return 0;
}
```

However, the underlying implementation rejects replacement:

```
function _setShieldVerifier(
    address verifier_
) internal {
    if (address(shieldVerifier) != address(0))
        revert ShieldVerifierAlreadySet();

    if (verifier_ == address(0))
        revert ZeroAddress();

    shieldVerifier = IVerifier(verifier_);

    emit ShieldVerifierSet(verifier_);
}
```

The prohibition appears intentional, as it prevents an already-configured verifier from being removed or replaced with an unsafe implementation.

IMPACT

Governance participants may schedule changes that cannot be executed, wasting transaction fees and introducing unnecessary administrative confusion.

Additionally, incorrectly configured shield verifiers cannot be replaced after initialization, potentially requiring deployment of a new pool.

RECOMMENDATION

Align the governance interface with the shield verifier's intended immutability.

Remove unsupported verifier replacement operations from the governance whitelist and clearly document that the shield verifier cannot be changed once initialized.

If future verifier rotation is required, establish a secure upgrade process that preserves the protocol's proof-validation guarantees.

REMEDIATION

ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. Harmless, and it will be removed in the next vault.

I-05 Expensive Zero-Knowledge Proof Verification Precedes Basic Deposit Validation

SEVERITY INFORMATIONAL STATUS ACKNOWLEDGED

LOCATION

ShieldPoolCore.sol - shieldBound(), _shield().

DESCRIPTION

The shieldBound() function performs zero-knowledge proof verification before executing several inexpensive deposit validation checks.

These checks include rejecting zero amounts, detecting duplicate commitments, and enforcing deposit limits.

Consequently, transactions that would inevitably fail basic validation still execute the comparatively expensive proof verification step before reverting.

This results in unnecessary gas expenditure for invalid deposit transactions.

VULNERABLE CODE

```
function shieldBound(
    address asset,
    uint256 amount,
    bytes32 commitment,
    bytes calldata proof
) external payable nonReentrant {
    if (address(shieldVerifier) == address(0))
        revert VerifierNotSet();

    uint256[] memory inputs = new uint256[](3);

    inputs[0] = uint256(commitment);
    inputs[1] = amount;
    inputs[2] = uint256(uint160(asset));

    if (!shieldVerifier.verify(proof, inputs))
        revert InvalidProof();

    _shield(asset, amount, commitment);
}
```

The basic checks execute afterward:

```
if (commitment == bytes32(0))
    revert ZeroCommitment();

if (amount == 0)
    revert InvalidAmount();

if (commitmentSeen[commitment])
    revert DuplicateCommitment();

if (depositLimitsOn) {
    DepositLimit memory lim = depositLimit[asset];

    if (
        amount > lim.perDeposit ||
        deposited[asset] + amount > lim.total
    ) {
        revert OverDepositLimit();
    }
}
```

IMPACT

Users submitting deposits that fail basic validation incur unnecessarily high transaction costs.

Although the issue does not directly compromise protocol funds, it reduces transaction efficiency and creates avoidable computational overhead.

RECOMMENDATION

Perform inexpensive validation checks before executing zero-knowledge proof verification.

Ensure that the protocol rejects obviously invalid deposit requests as early as possible, while maintaining the existing proof-verification security guarantees.

REMEDIATION ● ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. Gas only, on failed transactions. We'll reorder it in the next vault.

I-06 Merkle Root History Storage Introduces Additional Gas Overhead

SEVERITY ● INFORMATIONAL STATUS ● ACKNOWLEDGED

LOCATION

IncrementalMerkleTreePoseidon.sol - insert(), isKnownRoot().

DESCRIPTION

The Merkle tree implementation maintains a root-history ring buffer that is updated following each successful commitment insertion.

However, on-chain root verification relies on the separate knownRoots mapping instead of consulting the stored root-history buffer.

Consequently, additional persistent storage operations are performed even though the history buffer is not required for on-chain root validation.

The implementation documents that the buffer exists for off-chain observability, making this primarily an efficiency consideration rather than an unintended security flaw.

RELEVANT CODE

```
self.currentRoot = currentLevelHash;
self.knownRoots[currentLevelHash] = true;

uint32 newRootIndex =
    (self.currentRootIndex + 1)
    % self.rootHistorySize;

self.currentRootIndex = newRootIndex;

self.rootHistory[newRootIndex] =
    currentLevelHash;
```

On-chain root validation instead uses:

```
function isKnownRoot(
    Tree storage self,
    uint256 root
) internal view returns (bool) {
    if (root == 0) return false;

    return self.knownRoots[root];
}
```

IMPACT

Deposits and private transfers incur additional storage costs when commitments are inserted.

This overhead accumulates as the protocol processes more transactions.

No direct security impact has been identified.

RECOMMENDATION

Review whether the root-history storage is actively required by off-chain services or monitoring infrastructure.

If the data is unnecessary, consider removing redundant storage updates and relying on existing root-validation state and emitted events.

If external systems depend on the history buffer, retain it and document the resulting gas tradeoff.

REMEDIATION ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. Kept on purpose for off-chain use. Nothing reads it today, so we'll drop it in the next vault.

I-07 Missing Constructor Validation Allows Invalid Proof Verifier Dependencies

SEVERITY INFORMATIONAL STATUS ACKNOWLEDGED

LOCATION

DualProofVerifier.sol - constructor().

DESCRIPTION

The DualProofVerifier contract receives the addresses of the withdrawal and transfer proof verifiers during deployment.

However, the constructor stores these addresses without ensuring that they are nonzero or correspond to deployed contracts.

Consequently, the verifier router may be deployed with invalid dependencies, causing the corresponding proof-verification operations to fail.

Because the configured addresses are immutable, correcting the mistake requires deploying a new verifier router.

VULNERABLE CODE

```
contract DualProofVerifier is IVerifier {
    IVerifier public immutable unshieldVerifier;
    IVerifier public immutable transferVerifier;

    constructor(
        IVerifier unshieldVerifier_,
        IVerifier transferVerifier_
    ) {
        unshieldVerifier = unshieldVerifier_;
        transferVerifier = transferVerifier_;
    }
}
```

The verification function forwards requests to the configured dependencies:

```
function verify(
    bytes calldata proof,
    uint256[] calldata publicInputs
) external view returns (bool) {
    if (publicInputs.length == 5) {
        return unshieldVerifier.verify(
            proof,
            publicInputs
        );
    }

    if (publicInputs.length == 4) {
        return transferVerifier.verify(
            proof,
            publicInputs
        );
    }

    return false;
}
```

IMPACT

Incorrect deployment parameters may render private transfers or withdrawals unavailable when the affected verifier path is called.

Since the dependencies are immutable, recovery requires redeployment and administrative reconfiguration.

RECOMMENDATION

Validate verifier dependencies during deployment to ensure that the provided addresses are nonzero and reference deployed contracts.

Introduce deployment checks confirming that both verifier implementations are correctly configured and operational before enabling the associated pool.

REMEDIATION ● ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. The live verifiers are verified and tested. We'll add the checks at the next deploy.

I-08 ERC-20 tokens sent directly to the pool cannot be recovered

SEVERITY ● INFORMATIONAL STATUS ● ACKNOWLEDGED

LOCATION

contracts/src/ShieldPoolCore.sol#L309-L318.

DESCRIPTION

`sweepStrayNative` recovers native currency sent to the pool by mistake, but there is no equivalent for ERC-20 tokens. Tokens sent with a plain transfer (for example a user sending USDG to the vault address from an exchange instead of shielding) are not counted in `deposited` or `feesOwed`, and no function can move them. The Robinhood Chain and Tempo pools accept only ERC-20 assets, while the Base and Ethereum vaults also hold native ETH. The native-only rescue can recover surplus native ETH but cannot recover ERC-20 tokens sent directly to any of these vaults.

IMPACT

User funds sent by a common mistake are permanently lost. Pool accounting is not affected.

PROOF OF CONCEPT

```
function test_L03_stray_erc20_unrecoverable() public {
    _deposit(alice, PER, 1);
    vm.prank(bob);
    usd.transfer(address(pool), 1_000e6);
    assertEq(usd.balanceOf(address(pool)), PER + 1_000e6);
    assertEq(pool.deposited(address(usd)), PER);
    vm.expectRevert(ShieldPoolCore.NothingToSweep.selector);
    pool.sweepStrayNative(owner);
}
```

RECOMMENDATION

Generalise the sweep to any asset. Allow only `balanceOf(this) - deposited[asset] - feesOwed[asset]` to be moved, to a fixed address, keeping the same rule that funds backing notes are never touched.

REMEDIATION ● ACKNOWLEDGED

GLOAM TEAM

Status: Acknowledged. We'll add a limited ERC-20 sweep in the next vault.

09 - PLEASE READ

DISCLAIMER

A security audit can never guarantee the complete absence of vulnerabilities. Audits are a time, resource and expertise-bound effort in which we aim to identify as many issues as possible.

This review covers only the files listed in the Scope section at the audited commit. Code changed after that commit, external dependencies, deployment configuration and operational practices may introduce additional risk.

10 - WHO WE ARE

ABOUT KANN AUDITS

Kann Audits is a Web3 security audit company helping DeFi protocols, blockchain networks, and infrastructure teams build securely at every stage, from development to launch and beyond.

EXPERT SECURITY AUDITS

Human-led smart contract security audits built around the architecture, business logic, integrations and risk profile of complex Web3 protocols.

AI SECURITY SCAN

Kann AI extracts Solidity call graphs and returns function-level security signals for expert validation.

FORMAL VERIFICATION

Define security properties for critical protocol behaviour and evaluate whether the implementation satisfies those specifications.

INCIDENT RESPONSE

A direct triage path for suspected exploits, critical vulnerabilities, suspicious transactions and urgent remediation review.

KANN AUDITS

WEB3 SECURITY AUDITS

GET IN TOUCH

WEBSITE	kannaudits.com
PORTFOLIO	github.com/KannAudits
EMAIL	contact@kannaudits.com
X	@KannAudits
TELEGRAM	@KannAudits

© 2026 Kann Audits. Prepared for Gloam.