Smart Contract Security Audit

Document Purpose: Comprehensive security audit findings for ShieldedPool-MultiToken and related contracts
Audit Date: January 2025
Contract Version: V3 (Deployed), V4 (Pending with fixes)
Status: Complete audit report

Table of Contents

Executive Summary

Total Issues Found: 10
- **Critical:** 2 - **High:** 3 (1 legacy/deprecated) - **Medium:** 2 - **Low:** 1 - **Gas:** 2

Current Deployment Status: - **Deployed Contract:** V3 (`0x05D32B760ff49678FD833a4E2AbD516586362b`
- **Status:** Does NOT include security fixes

External Dependencies: - 7 External Contracts Called - 3 External Systems

See `SMART_CONTRACT_AUDIT_FIXES.md` for implemented fixes.

External Dependencies

External Contracts Called

1. **Hasher** (`0x1931f2D78930f5c3b0ce65d27F56F35Fa4fdA67D`)
   - Purpose: MiMC Sponge hash function implementation
   - Risk: Trusted for commitment/nullifier generation
   - Status: Deployed, immutable
2. **ShieldVerifier** (`0x7b5259d3d048195e4b670B87DB4dac0196a132Da`)
   - Purpose: Shield proof verification (Groth16)
   - Risk: Trusted for shield operation security
   - Status: Deployed, immutable
3. **TransferVerifier** (`0x10587baC7846D926938D2AeB432860bc6791C596`)

- Purpose: Transfer proof verification (Groth16)
- Risk: Trusted for transfer operation security
- Status: Deployed, immutable

4. **UnshieldVerifier** (`0xBaEDFC77a7dA1e36D4566d646c48F9359015DB4E`)
   - Purpose: Unshield proof verification (Groth16)
   - Risk: Trusted for unshield operation security
   - Status: Deployed, immutable
5. **SwapVerifier** (`0xcAa64148b6789183cB67A7BaFC6cE37F46B363cc`)
   - Purpose: Swap proof verification (Groth16)
   - Risk: Trusted for swap operation security
   - Status: Deployed, immutable
6. **ERC20 Tokens** (Various addresses)
   - Purpose: Token deposits/withdrawals
   - Risk: Trusted for token standard compliance
   - Status: External tokens, cannot guarantee behavior
7. **Platform Treasury** (`0xdFc15203f5397495Dada3D7257Eed1b00DCFF548`)
   - Purpose: Receives platform fees from swaps
   - Risk: Centralized address, must be trusted
   - Status: Configured, owner-controlled

External Systems

1. **Zero-Knowledge Proof System**
   - Purpose: Privacy-preserving proof generation
   - Risk: Circuit correctness, proof generation security
   - Status: Frontend-generated proofs, trust in circuit compilation
2. **DEX Router** (Currently unused - `address(0)`)
   - Purpose: Swap rate quotes (planned for production)
   - Risk: Trusted for accurate swap rates
   - Status: Not yet integrated
3. **Relayer Service**
   - Purpose: Gasless transactions (pays gas for users)
   - Risk: Relayer must be trusted not to censor transactions
   - Status: Centralized service, operational risk

Critical Severity Issues

Issue #1: ShieldedPoolMultiToken.sol - Swap Function Allows Draining Pool Through Liquidity Check Bypass

Severity: CRITICAL
Status: Pending Fix (See `SMART_CONTRACT_AUDIT_FIXES.md`)

Vulnerability Description The swap function checks liquidity **AFTER** verifying the proof but uses the proof's `_outputAmount` parameter directly with-

out validating it matches actual swap rates. An attacker can generate a valid proof with a very small `_outputAmount` (e.g., 1 wei) while swapping a large input amount. Since the liquidity check only validates `totalOutputNeeded = _outputAmount + _platformFee`, and the contract uses the proof's outputAmount directly without calling `_executeSwap()`, an attacker can drain valuable tokens by swapping worthless tokens for valuable ones at an artificially favorable rate.

Impact An attacker can drain the entire pool of valuable tokens by repeatedly swapping worthless tokens at artificially favorable exchange rates. For example, swapping 1 wei of a worthless token for large amounts of WETH or stablecoins, limited only by the pool's balance.

Attack Scenario

```
// Attacker has a shielded note of 1000 worthless tokens (e.g., FAKE_TOKEN)
// Pool has 100 WETH

// Step 1: Generate proof with manipulated exchange rate
// Input: 1000 FAKE_TOKEN
// Output: 50 WETH (proof claims this is the correct rate)
proof = generateSwapProof(
    inputAmount: 1000e18,
    outputAmount: 50e18, // Artificially high output
    tokenIn: FAKE_TOKEN,
    tokenOut: WETH
);

// Step 2: Call swap with the malicious proof
pool.swap(
    proof,
    root,
    inputNullifier,
    outputCommitment1,
    outputCommitment2,
    FAKE_TOKEN,
    WETH,
    1000e18, // swapAmount
    50e18,   // outputAmount (from malicious proof)
    0.001e18, // small platform fee
    1e18,    // minAmountOut (easily satisfied)
    encryptedMemo
);

// Result: Attacker gets 50 WETH for worthless tokens
// Repeat to drain pool
```

Affected Code

```
function swap(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _inputNullifier,
    bytes32 _outputCommitment1,
    bytes32 _outputCommitment2,
    address _tokenIn,
    address _tokenOut,
    uint256 _swapAmount,
    uint256 _outputAmount, // Attacker controls this via proof
    uint256 _platformFee,
    uint256 _minAmountOut,
    bytes calldata _encryptedMemo
) external nonReentrant {
    // ...

    // Calculate total output needed (user's net amount + platform fee)
    uint256 totalOutputNeeded = _outputAmount + _platformFee;

    // Check liquidity - but _outputAmount comes from attacker's proof!
    if (_tokenOut == NATIVE_TOKEN) {
        if (address(this).balance < totalOutputNeeded) revert InsufficientPoolBalance();
    } else {
        uint256 availableBalance = IERC20(_tokenOut).balanceOf(address(this));
        if (availableBalance < totalOutputNeeded) revert InsufficientPoolBalance();
    }

    // Verify proof - but proof can have any outputAmount!
    uint256[8] memory publicInputs = [
        uint256(_root),
        uint256(_inputNullifier),
        uint256(_outputCommitment1),
        uint256(_outputCommitment2),
        tokenInUint,
        tokenOutUint,
        _swapAmount,
        _outputAmount  // Attacker's chosen amount
    ];

    // ...

    // Use the proof's outputAmount directly (source of truth for MVP/testnet)
    uint256 finalAmountOut = _outputAmount;
```

```
    // Note: _executeSwap() is not called in MVP mode
    // The proof's outputAmount is used directly!
}
```

Proposed Fix See `SMART_CONTRACT_AUDIT_FIXES.md` for implementation details.

Issue #2: ShieldedPoolMultiToken.sol - Centralized Owner Can Rug Pull by Removing Token Support

Severity: CRITICAL
Status: Pending Fix (See `SMART_CONTRACT_AUDIT_FIXES.md`)

Vulnerability Description The owner can remove support for any ERC20 token at any time via `removeSupportedToken()`. This allows the owner to trap user funds by: 1) Waiting for users to shield tokens, 2) Removing token support, 3) Preventing users from unshielding their tokens since `unshieldToken()` checks `supportedTokens[_token]`. While native DOGE cannot be removed, all ERC20 tokens are vulnerable. This creates a honeypot risk where users' shielded ERC20 tokens become permanently locked.

Impact Users' ERC20 tokens can be permanently locked in the contract if the owner removes token support after users have shielded their tokens. This could affect all users holding shielded ERC20 tokens, resulting in complete loss of funds for those tokens.

Attack Scenario

```
// Step 1: Owner adds USDC support
owner.addSupportedToken(USDC);

// Step 2: User shields 1000 USDC
user.shieldToken(USDC, 1000e18, commitment);

// Step 3: Owner removes USDC support (rug pull)
owner.removeSupportedToken(USDC);

// Step 4: User tries to unshield but fails
user.unshieldToken(proof, root, nullifier, recipient, USDC, 1000e18, relayer, fee);
// Reverts with UnsupportedToken error
// User's 1000 USDC is now permanently locked
```

Affected Code

```
function removeSupportedToken(address _token) external onlyOwner {
    if (_token == NATIVE_TOKEN) revert Unauthorized(); // Can't remove native
    supportedTokens[_token] = false;
    emit TokenRemoved(_token);
}

function unshieldToken(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    address _recipient,
    address _token,
    uint256 _amount,
    address _relayer,
    uint256 _fee
) external nonReentrant {
    if (!supportedTokens[_token]) revert UnsupportedToken(); // Blocks unshielding
    _unshield(_proof, _root, _nullifierHash, _recipient, _token, _amount, _relayer, _fee);
}
```

Proposed Fix See `SMART_CONTRACT_AUDIT_FIXES.md` for implementation details.

High Severity Issues

Issue #3: DogeRouter.sol - DogeRouter Vulnerable to Proof Manipulation Attack

Severity: HIGH (Legacy/Deprecated)
Status: Deprecated - Not Used by ShieldedPoolMultiToken V3
Priority: LOW - Legacy contract only

Note: DogeRouter is legacy infrastructure for old fixed-denomination MixerPool contracts. Current ShieldedPoolMultiToken V3 handles native DOGE directly (`shieldNative()` and `unshieldNative()` functions) and does not use DogeRouter. This vulnerability does not affect ShieldedPoolMultiToken V3.

Recommendation: If legacy MixerPool pools are no longer in active use, DogeRouter can be considered deprecated. If legacy pools are still active, this vulnerability should be fixed or users should migrate to ShieldedPoolMultiToken V3.

Vulnerability Description The DogeRouter's `withdrawDoge` function expects proofs to be generated with the router as recipient, but there's no on-chain verification that ensures this. An attacker can generate a proof with themselves

as the recipient and call `withdrawDoge`. The mixer pool will send wDOGE directly to the attacker instead of the router, bypassing the router entirely. The router will then try to unwrap a zero balance, and the subsequent native DOGE transfer will fail or send 0 ETH.

Impact Attackers can steal wDOGE from the mixer pool by generating proofs with themselves as recipients while calling the router's withdrawDoge function. The router cannot detect this manipulation and will process a withdrawal of 0 tokens while the attacker receives the actual funds.

Attack Scenario

```
// Step 1: Attacker generates a withdrawal proof with THEMSELVES as recipient
proof = generateWithdrawProof(
    recipient: attackerAddress, // NOT the router!
    amount: 100e18,
    nullifier: nullifier
);

// Step 2: Attacker calls DogeRouter.withdrawDoge
router.withdrawDoge(
    pool,
    proof,
    root,
    nullifierHash,
    dummyRecipient, // Doesn't matter, will receive 0
    address(0),
    0
);

// Step 3: What happens:
// - MixerPool.withdraw sends 100 wDOGE to attackerAddress (from proof)
// - Router's balance doesn't change (balanceBefore == balanceAfter)
// - Router tries to unwrap 0 wDOGE
// - dummyRecipient receives 0 native DOGE
// - Attacker keeps 100 wDOGE
```

Affected Code

```
function withdrawDoge(
    address pool,
    uint256[8] calldata proof,
    bytes32 root,
    bytes32 nullifierHash,
    address payable recipient,
    address payable relayer,
```

```
    uint256 fee
) external nonReentrant {
    if (!validPools[pool]) revert InvalidPool();

    IMixerPool mixerPool = IMixerPool(pool);

    // Get wDOGE balance before withdrawal
    uint256 balanceBefore = wdoge.balanceOf(address(this));

    // Withdraw wDOGE to this contract
    // Note: The proof must be generated with THIS CONTRACT as the recipient
    // BUT: There's no way to verify this on-chain!
    mixerPool.withdraw(
        proof,
        root,
        nullifierHash,
        address(this), // Router expects to receive funds here
        relayer,
        fee
    );

    // Calculate received amount
    uint256 balanceAfter = wdoge.balanceOf(address(this));
    uint256 received = balanceAfter - balanceBefore; // Will be 0 if proof had different rec

    // Unwrap wDOGE to native DOGE
    wdoge.withdraw(received); // Withdraws 0 if attacker redirected funds
}
```

Proposed Fix Option 1: Remove DogeRouter and handle wrapping/unwrapping in main pool

Option 2: Use a commit-reveal scheme:

```
contract DogeRouter {
    mapping(bytes32 => bytes32) public pendingWithdrawals;

    function initiateWithdrawal(bytes32 withdrawalHash) external {
        pendingWithdrawals[withdrawalHash] = keccak256(abi.encodePacked(msg.sender, block.ti
    }

    function completeWithdrawal(
        address pool,
        uint256[8] calldata proof,
        bytes32 root,
        bytes32 nullifierHash,
```

```
        address payable recipient,
        address payable relayer,
        uint256 fee
    ) external nonReentrant {
        bytes32 withdrawalHash = keccak256(abi.encodePacked(nullifierHash, recipient));
        require(pendingWithdrawals[withdrawalHash] != bytes32(0), "Withdrawal not initiated'
        delete pendingWithdrawals[withdrawalHash];

        // Now proceed with withdrawal knowing it was pre-authorized
        // ...
    }
}
```

Option 3: Modify MixerPool to validate recipient matches msg.sender for router calls

---

Issue #4: ShieldedPoolMultiToken.sol - Platform Fee Bypass Through Zero Platform Fee Input

Severity: HIGH
Status: Pending Review

Vulnerability Description The swap function accepts `_platformFee` as an input parameter without validating it equals the expected 5 DOGE equivalent. An attacker can call swap with `_platformFee = 0` or any small value. The contract will only send the specified platform fee to treasury, allowing users to bypass the intended 5 DOGE fee. There's no on-chain validation that the platform fee matches the expected amount.

Impact Protocol loses expected revenue from swap operations. Users can perform swaps without paying the intended 5 DOGE platform fee, undermining the protocol's economic model and sustainability.

Attack Scenario

```
// Expected behavior: User pays 5 DOGE equivalent fee per swap
// Actual behavior: User can set fee to 0

// Generate proof for swap
proof = generateSwapProof(...);

// Call swap with 0 platform fee
pool.swap(
    proof,
    root,
```

```
    inputNullifier,
    outputCommitment1,
    outputCommitment2,
    tokenIn,
    tokenOut,
    swapAmount,
    outputAmount,
    0, // _platformFee set to 0 instead of 5 DOGE equivalent!
    minAmountOut,
    encryptedMemo
);

// Result: Swap executes successfully with no fee paid to treasury
```

Affected Code

```
function swap(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _inputNullifier,
    bytes32 _outputCommitment1,
    bytes32 _outputCommitment2,
    address _tokenIn,
    address _tokenOut,
    uint256 _swapAmount,
    uint256 _outputAmount,
    uint256 _platformFee, // User controls this!
    uint256 _minAmountOut,
    bytes calldata _encryptedMemo
) external nonReentrant {
    // ...

    // Send platform fee to treasury (if > 0)
    // But there's no validation that _platformFee equals expected 5 DOGE!
    if (_platformFee > 0) {
        if (_tokenOut == NATIVE_TOKEN) {
            (bool success, ) = PLATFORM_TREASURY.call{value: _platformFee}("");
            if (!success) revert("Failed to send platform fee to treasury");
        } else {
            IERC20(_tokenOut).safeTransfer(PLATFORM_TREASURY, _platformFee);
        }
    }
    // User can set _platformFee = 0 to avoid fees entirely!
}
```

Proposed Fix Option 1: Add a constant for the expected platform fee

```
uint256 public constant PLATFORM_FEE_DOGE = 5e18; // 5 DOGE

function swap(...) external nonReentrant {
    // Calculate expected platform fee in output token
    uint256 expectedPlatformFee;
    if (_tokenOut == NATIVE_TOKEN) {
        expectedPlatformFee = PLATFORM_FEE_DOGE;
    } else {
        // Get DOGE/tokenOut exchange rate and calculate fee
        expectedPlatformFee = _convertDogeToToken(PLATFORM_FEE_DOGE, _tokenOut);
    }

    // Validate provided platform fee matches expected
    if (_platformFee < expectedPlatformFee) {
        revert InsufficientPlatformFee();
    }

    // Or simply remove _platformFee parameter and calculate it internally
    // This ensures the fee cannot be manipulated
}
```

Option 2: Remove `_platformFee` parameter and calculate it internally

Issue #5: ShieldedPoolMultiToken.sol - Missing Validation of Change Commitment in Unshield Function

Severity: HIGH
Status: Pending Review

Vulnerability Description The `_unshield` function passes a hardcoded `uint256(0)` for the changeCommitment parameter when calling the verifier, indicating V1 doesn't support change. However, the UnshieldVerifierV3 contract expects 7 parameters including changeCommitment. If a user generates a proof with a non-zero changeCommitment expecting to receive change, the contract ignores it and doesn't insert the change commitment into the tree, causing permanent loss of the change amount.

Impact Users who generate partial unshield proofs expecting change will lose the change amount permanently. For example, if a user has a 100 DOGE note and unshields 30 DOGE expecting 70 DOGE change, they will receive 30 DOGE but the 70 DOGE change is lost forever.

Attack Scenario

```
// User has a shielded note of 100 DOGE
// User wants to unshield 30 DOGE and keep 70 DOGE shielded

// User generates proof with change commitment
changeCommitment = generateCommitment(70 DOGE);
proof = generatePartialUnshieldProof(
    amount: 30 DOGE,
    changeCommitment: changeCommitment,
    changeAmount: 70 DOGE
);

// User calls unshield
pool.unshieldNative(
    proof,
    root,
    nullifierHash,
    recipient,
    30 DOGE, // amount to unshield
    relayer,
    fee
);

// Result:
// - User receives 30 DOGE
// - Change commitment is ignored (hardcoded to 0)
// - 70 DOGE change is lost forever
```

Affected Code

```
function _unshield(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    address _recipient,
    address _token,
    uint256 _amount,
    address _relayer,
    uint256 _fee
) internal {
    // ...

    // Verify proof with fixed-size array
    if (!unshieldVerifier.verifyProof(
        [_proof[0], _proof[1]],
```

```
        [[_proof[2], _proof[3]], [_proof[4], _proof[5]]],
        [_proof[6], _proof[7]],
        [
            uint256(_root),
            uint256(_nullifierHash),
            uint256(uint160(_recipient)),
            _amount,
            uint256(0),  // changeCommitment (V1 doesn't support change, use 0)
            uint256(uint160(_relayer)),
            _fee
        ]
    )) {
        revert InvalidProof();
    }

    // No logic to handle change commitment if provided!
    // Change amount would be lost
}
```

Proposed Fix Add support for partial unshield with change:

```
function _unshield(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    address _recipient,
    address _token,
    uint256 _amount,
    bytes32 _changeCommitment, // Add parameter
    address _relayer,
    uint256 _fee
) internal {
    // Verify proof with change commitment
    if (!unshieldVerifier.verifyProof(
        [_proof[0], _proof[1]],
        [[_proof[2], _proof[3]], [_proof[4], _proof[5]]],
        [_proof[6], _proof[7]],
        [
            uint256(_root),
            uint256(_nullifierHash),
            uint256(uint160(_recipient)),
            _amount,
            uint256(_changeCommitment), // Use actual change commitment
            uint256(uint160(_relayer)),
            _fee
        ]
```

```
    )) {
        revert InvalidProof();
    }

    // Insert change commitment if non-zero
    if (_changeCommitment != bytes32(0)) {
        _insert(_changeCommitment);
        commitments[_changeCommitment] = true;
    }

    // ... rest of function
}
```

Note: This requires updating the `unshieldToken()` and `unshieldNative()` functions to accept and pass the change commitment.

Medium Severity Issues

Issue #6: MerkleTreeWithHistory.sol - Merkle Tree Root Manipulation Through Rapid Insertions

Severity: MEDIUM
Status: Pending Review

Vulnerability Description The Merkle tree maintains a circular buffer of only 30 historical roots. An attacker can rapidly insert 30 new leaves (via shield operations) to cycle through all root slots, effectively invalidating any pending proofs that rely on older roots. This can be used to grief users who have generated proofs but haven't submitted them yet, or to invalidate specific roots that the attacker knows are being used for high-value withdrawals.

Impact Users' pending transactions can be invalidated, causing denial of service. High-value withdrawals can be specifically targeted if the attacker knows which root they're using. Users would need to regenerate proofs with new roots, potentially losing gas fees and facing delays.

Attack Scenario

```
// Step 1: User generates a withdrawal proof with current root
root_old = pool.getLatestRoot();
proof = generateUnshieldProof(root_old, ...);

// Step 2: Attacker rapidly shields 30 times to cycle roots
for (uint i = 0; i < 30; i++) {
    bytes32 commitment = keccak256(abi.encodePacked(i, block.timestamp));
```

```
    pool.shieldNative{value: 0.001 ether}(commitment);
}

// Step 3: User's proof is now invalid
pool.unshieldNative(proof, root_old, ...); // Reverts: InvalidProof (root not known)

// User must regenerate proof with new root, losing gas and time
```

Affected Code

```
function _insert(bytes32 leaf) internal returns (uint256 leafIndex) {
    uint256 _nextLeafIndex = nextLeafIndex;

    if (_nextLeafIndex >= 2 ** TREE_DEPTH) {
        revert MerkleTreeFull();
    }

    // ... tree update logic ...

    // Update root history - circular buffer of only 30 roots!
    uint256 newRootIndex = (currentRootIndex + 1) % ROOT_HISTORY_SIZE;
    roots[newRootIndex] = currentLevelHash; // Overwrites old root at this position
    currentRootIndex = newRootIndex;

    nextLeafIndex = _nextLeafIndex + 1;
    leafIndex = _nextLeafIndex;

    emit LeafInserted(leaf, leafIndex, currentLevelHash);
}

function isKnownRoot(bytes32 root) public view returns (bool) {
    if (root == bytes32(0)) {
        return false;
    }

    uint256 i = currentRootIndex;
    do {
        if (root == roots[i]) {
            return true;
        }
        if (i == 0) {
            i = ROOT_HISTORY_SIZE - 1;
        } else {
            i--;
        }
    } while (i != currentRootIndex);
```

```
    return false; // Root no longer in history!
}
```

Proposed Fix Option 1: Increase root history size significantly

```
uint256 public constant ROOT_HISTORY_SIZE = 500; // Was 30
```

Option 2: Implement a time-based root expiry

```
mapping(bytes32 => uint256) public rootTimestamps;
uint256 public constant ROOT_EXPIRY_TIME = 1 days;

function _insert(bytes32 leaf) internal returns (uint256 leafIndex) {
    // ... existing logic ...

    // Store timestamp with root
    rootTimestamps[currentLevelHash] = block.timestamp;

    // ... rest of function ...
}

function isKnownRoot(bytes32 root) public view returns (bool) {
    // Check if root exists and hasn't expired
    if (rootTimestamps[root] == 0) return false;
    return block.timestamp <= rootTimestamps[root] + ROOT_EXPIRY_TIME;
}
```

Issue #7: ShieldedPoolMultiToken.sol - Commitment Existence Check Can Be Bypassed in Transfer Function

Severity: MEDIUM
Status: Pending Review

Vulnerability Description The transfer function adds output commitments to the `commitments` mapping after inserting them into the Merkle tree, but it doesn't check if these commitments already exist before insertion. While `_insert` will add duplicate commitments to different leaf positions in the tree, this breaks the uniqueness assumption and could allow an attacker to create multiple notes with the same commitment, potentially enabling double-spending in specific scenarios where commitment uniqueness is assumed by the ZK circuits.

Impact Commitment uniqueness assumption is broken. If the same commitment exists at multiple leaf positions, it could potentially be spent multiple times if the ZK circuit doesn't properly validate the Merkle path uniqueness. This could lead to inflation of the token supply within the shielded pool.

Attack Scenario

```
// Step 1: User A shields with commitment C1
pool.shieldNative{value: 10 ether}(C1);
// C1 is at leaf index 0

// Step 2: User B generates a transfer proof that outputs C1 again
// (This shouldn't be possible with proper ZK circuit, but if circuit has bugs...)
transferProof = generateTransferProof(
    outputCommitment1: C1, // Same commitment!
    outputCommitment2: 0
);

pool.transfer(
    transferProof,
    root,
    nullifier,
    C1, // Duplicate commitment
    0,
    relayer,
    fee,
    memo1,
    memo2
);

// Result: C1 now exists at leaf index 0 AND leaf index 1
// Both positions could potentially be spent if circuit allows
```

Affected Code

```
function transfer(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    bytes32 _outputCommitment1,
    bytes32 _outputCommitment2,
    address _relayer,
    uint256 _fee,
    bytes calldata _encryptedMemo1,
    bytes calldata _encryptedMemo2
) external nonReentrant {
    // ... verification ...

    // Insert new commitments WITHOUT checking if they already exist!
    uint256 leafIndex1 = _insert(_outputCommitment1);
    commitments[_outputCommitment1] = true; // Set after insert
```

```
    uint256 leafIndex2 = 0;
    if (_outputCommitment2 != bytes32(0)) {
        leafIndex2 = _insert(_outputCommitment2);
        commitments[_outputCommitment2] = true; // Set after insert
    }

    // If commitment already existed, it's now in tree twice!
}
```

Proposed Fix

```
function transfer(
    uint256[8] calldata _proof,
    bytes32 _root,
    bytes32 _nullifierHash,
    bytes32 _outputCommitment1,
    bytes32 _outputCommitment2,
    address _relayer,
    uint256 _fee,
    bytes calldata _encryptedMemo1,
    bytes calldata _encryptedMemo2
) external nonReentrant {
    if (nullifierHashes[_nullifierHash]) revert NullifierAlreadySpent();
    if (!isKnownRoot(_root)) revert InvalidProof();

    // Check commitment uniqueness BEFORE insertion
    if (commitments[_outputCommitment1]) revert CommitmentAlreadyExists();
    if (_outputCommitment2 != bytes32(0) && commitments[_outputCommitment2]) {
        revert CommitmentAlreadyExists();
    }

    // ... verify proof ...

    // Mark commitments as existing BEFORE insertion
    commitments[_outputCommitment1] = true;
    uint256 leafIndex1 = _insert(_outputCommitment1);

    uint256 leafIndex2 = 0;
    if (_outputCommitment2 != bytes32(0)) {
        commitments[_outputCommitment2] = true;
        leafIndex2 = _insert(_outputCommitment2);
    }

    // ... rest of function
}
```

Low Severity Issues

Issue #8: ShieldVerifier.sol - Verifier Contracts Vulnerable to Malleability Attack

Severity: LOW
Status: Pending Review

Vulnerability Description The Groth16 verifier contracts don't implement proper proof malleability checks. While Groth16 proofs are generally non-malleable, the verifier accepts proofs where point coordinates could potentially be replaced with equivalent points on the curve. An attacker could potentially modify proof elements (particularly the G1 points) to create a different but still valid proof for the same public inputs, which could bypass replay protection mechanisms that rely on proof uniqueness.

Impact If an attacker can create malleable proofs, they could potentially bypass certain application-level checks that assume proof uniqueness. While the nullifier mechanism prevents double-spending, other security assumptions about proof uniqueness could be violated.

Attack Scenario

```
// Theoretical attack (requires deep understanding of curve operations):
// Given a valid proof (pA, pB, pC) for public inputs P
// Attacker could potentially create (pA', pB', pC') where:
// - pA' = (pA.x, -pA.y mod q) // Negated y-coordinate
// - Adjust pB' and pC' accordingly
// - Both proofs verify for the same public inputs

// This could bypass mechanisms that track "used proofs" by proof hash
// Rather than tracking used nullifiers
```

Affected Code

```
function verifyProof(uint[2] calldata _pA, uint[2][2] calldata _pB, uint[2] calldata _pC, ui
    assembly {
        function checkField(v) {
            if iszero(lt(v, r)) {
                mstore(0, 0)
                return(0, 0x20)
            }
        }

        // Only checks if values are in field, not if points are canonical
```

```
        checkField(calldataload(add(_pubSignals, 0)))
        checkField(calldataload(add(_pubSignals, 32)))

        // No check for point malleability
        // Could accept (x, y) or (x, -y) as valid

        let isValid := checkPairing(_pA, _pB, _pC, _pubSignals, pMem)
        mstore(0, isValid)
        return(0, 0x20)
    }
}
```

Proposed Fix Add canonical point validation:

```
function verifyProof(...) public view returns (bool) {
    assembly {
        function checkCanonicalPoint(x, y) {
            // Ensure point is in canonical form
            // For BN254, ensure y < (q-1)/2 for canonical representation
            let halfQ := div(q, 2)
            if gt(y, halfQ) {
                mstore(0, 0)
                return(0, 0x20)
            }
        }

        // Check all points are canonical
        checkCanonicalPoint(calldataload(_pA), calldataload(add(_pA, 32)))
        checkCanonicalPoint(calldataload(_pC), calldataload(add(_pC, 32)))

        // ... rest of verification
    }
}
```

Or better: Use nullifiers for replay protection, not proof hashes (which the contract already does correctly)

Gas Optimization Issues

Issue #9: Hasher.sol - Hasher Contract Uses Inefficient MiMC Implementation

Severity: GAS
Status: Pending Optimization

Vulnerability Description The Hasher contract implements MiMC Sponge with 220 rounds, performing 440 iterations total (220 for first absorption, 220 for second). Each iteration involves multiple field operations (addmod, mulmod). This is extremely gas-intensive, costing approximately 200,000-300,000 gas per hash operation. Every Merkle tree insertion requires multiple hash operations (one per tree level), making shield/transfer/unshield operations very expensive.

Impact Users pay excessive gas fees for all operations. A single shield operation with 20 tree levels could cost 2-4 million gas. At 100 gwei gas price: 0.2-0.4 ETH per shield operation! This makes the protocol expensive to use and could price out many users, especially during high gas price periods.

Affected Code

```
function MiMCSponge(uint256 left, uint256 right) external view returns (uint256 result) {
    uint256 k = 0;
    uint256 t;
    uint256 currentState = left;

    // First absorption - 220 rounds
    for (uint256 i = 0; i < NUM_ROUNDS; i++) {
        t = addmod(addmod(currentState, roundConstants[i], FIELD_SIZE), k, FIELD_SIZE);
        t = mulmod(t, t, FIELD_SIZE); // t^2
        t = mulmod(t, t, FIELD_SIZE); // t^4
        currentState = mulmod(t, addmod(addmod(currentState, roundConstants[i], FIELD_SIZE),
    }
    currentState = addmod(currentState, k, FIELD_SIZE);

    currentState = addmod(currentState, right, FIELD_SIZE);

    // Second permutation - another 220 rounds!
    for (uint256 i = 0; i < NUM_ROUNDS; i++) {
        t = addmod(addmod(currentState, roundConstants[i], FIELD_SIZE), k, FIELD_SIZE);
        t = mulmod(t, t, FIELD_SIZE);
        t = mulmod(t, t, FIELD_SIZE);
        currentState = mulmod(t, addmod(addmod(currentState, roundConstants[i], FIELD_SIZE),
    }

    result = addmod(currentState, k, FIELD_SIZE);
}
```

Proposed Fix **Option 1: Use circomlibjs generated MiMC contract** (Recommended) - Optimized assembly implementation - This is mentioned in the comments but not implemented - Can reduce gas cost by 30-50%

Option 2: Reduce rounds for testnet (Not recommended for mainnet)

```
uint256 constant NUM_ROUNDS = 110; // Half the rounds for testing
```

Option 3: Implement assembly optimizations

```
function MiMCSponge(uint256 left, uint256 right) external view returns (uint256) {
    assembly {
        // Implement MiMC in assembly for gas optimization
        // This can reduce gas cost by 30-50%
    }
}
```

Option 4: Use a different ZK-friendly hash like Poseidon (Future) - Poseidon is more efficient than MiMC while maintaining security - Requires circuit regeneration

---

Issue #10: HasherAdapter.sol - HasherAdapter Double External Call Inefficiency

Severity: GAS
Status: Pending Optimization

Vulnerability Description The HasherAdapter's `hash` function makes an unnecessary external call to its own `MiMCSponge` function via `this.MiMCSponge(input, input)`. This creates an additional external call overhead (approximately 2,600 gas) on top of the already expensive MiMC operation. Since `hash` is frequently called for nullifier generation, this adds unnecessary gas costs to every operation.

Impact Every nullifier hash operation costs an additional ~2,600 gas due to the external call overhead. Over thousands of transactions, this results in significant unnecessary gas costs for users.

Affected Code

```
function hash(uint256 input) external view returns (uint256) {
    return this.MiMCSponge(input, input); // Unnecessary external call to self!
}

// Should be:
function hash(uint256 input) external view returns (uint256) {
    // Call MiMCSponge directly without 'this'
    uint256 k = 0;
    (uint256 xL, uint256 xR) = mimcSponge.MiMCSponge(input, 0, k);
    (xL, ) = mimcSponge.MiMCSponge(addmod(input, xL, FIELD_SIZE), xR, k);
    return xL;
}
```

Proposed Fix

```
contract HasherAdapter {
    IMiMCSponge public immutable mimcSponge;
    uint256 constant FIELD_SIZE = 21888242871839275222246405745257275088548364400416034343698

    constructor(address _mimcSponge) {
        mimcSponge = IMiMCSponge(_mimcSponge);
    }

    function MiMCSponge(uint256 left, uint256 right) external view returns (uint256) {
        uint256 k = 0;
        (uint256 xL, uint256 xR) = mimcSponge.MiMCSponge(left, 0, k);
        (xL, ) = mimcSponge.MiMCSponge(addmod(right, xL, FIELD_SIZE), xR, k);
        return xL;
    }

    function hash(uint256 input) external view returns (uint256) {
        // Directly implement the logic instead of calling this.MiMCSponge
        uint256 k = 0;
        (uint256 xL, uint256 xR) = mimcSponge.MiMCSponge(input, 0, k);
        (xL, ) = mimcSponge.MiMCSponge(addmod(input, xL, FIELD_SIZE), xR, k);
        return xL;
    }
}
```

Last Updated: January 2025
Document Version: 1.0
Next Review: After fixes are implemented

For implemented fixes, see: `SMART_CONTRACT_AUDIT_FIXES.md`