Sherpa Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

MrPotatoMagic

November 23, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Sherpa is a cross-chain yield system composed of a per-chain SherpaVault (ERC20 shares) and a chain-local SherpaUSD (a 6-decimals wrapper for USDC). Users deposit via the vault; funds are held as SherpaUSD in the vault. Deposits land as pending for the current round; on round rollover, pending converts to vault shares at the newly posted pricePerShare. An operator posts global price each round on the primary chain computes which is then replicated to the secondary chains. Withdrawals can be two-step via the wrapper or direct if "independence" is enabled. Cross-chain rebalancing is performed when there are large supply differences between chains. It's done by minting/burning SherpaUSD to/from vaults and then adjusting vault accounting with explicit, per-vault approvals.

4.1 Actors and Roles

- **1. Actors:**
  - **Users:** Deposit USDC (through SherpaUSD via the vault), hold vault shares, redeem shares from receipts, unstake to withdraw SherpaUSD, or request epoch withdrawals.
  - **Operator:** Runs daily operations: round rolls, applies global price, pauses/unpauses, performs rebalancing adjustments, processes wrapper withdrawals. Custodies funds (USDC) from SherpaUSD to run yield strategies.
  - **Keeper (SherpaUSD):** Authorized vault address to pull user USDC and to perform permissioned wrapper mints/burns tied to vault operations.
  - **Owner:** Sets operator/keeper, toggles config (caps, independence, deposits), can rescue non-wrapper tokens from the vault, set stable wrapper, and manage CCIP pool allowlist.
- **2. Roles/Checks:**
  - **SherpaVault** `onlyOperator`: `rollToNextRound`, `applyGlobalPrice`, `setSystemPaused`, `adjustTotalStaked`, `adjustAccountingSupply`, `processWrapperWithdrawals`.
  - **SherpaVault** `onlyOwner`: `setPrimaryChain`, `setCap`, `setDepositsEnabled`, `setAllowIndependence`, `setStableWrapper`, CCIP pool add/remove, rescue tokens (not the wrapper token).

- **SherpaUSD** `onlyKeeper`: `depositToVault`, `permissionedMint/Burn`, `processWithdrawals`, `initiateWithdrawalFromVault`.
- **SherpaUSD** `onlyOperator`: `ownerMint/Burn` (rebal approvals), `transferAsset`.
- **SherpaUSD** `onlyOwner`: `setOperator`, `setKeeper`, `setAutoTransfer`.

4.2 Key Components

- **SherpaVault:**
  - **Shares (ERC20):** Represent a claim on SherpaUSD (wrapped USDC) held by the vault.
  - **Rounds & Pricing:** `roundPricePerShare[round]` is set each roll (round N finalizes price for N-1). Pending deposits convert to shares at roll.
  - **Accounting:**
    - `totalStaked` (SherpaUSD held by vault),
    - `accountingSupply` (logical supply of shares, immune to CCIP mint/burn),
    - `vaultState.totalPending` (current-round deposits not yet converted).
  - **Receipts:** Track user pending amount + unredeemed shares until redeemed.
  - **Pausing:** `setSystemPaused` blocks user flows and auto-unpauses after 24h.
- **SherpaUSD (USDC wrapper):**
  - **Epoch Withdrawals:** Users burn SherpaUSD to create receipts; completion is possible after epoch increments (keeper runs `processWithdrawals` to move to next epoch).
  - **Operator Rebalancing:** `ownerMint/Burn(to/from vault)` sets per-vault approvals (`approvedTotalStakedAdjustment`, `approvedAccountingAdjustment`). Vault later consumes these via `adjustTotalStaked` and `adjustAccountingSupply`, the latter validates the share math against the last price.

4.3 Core Flows

- **Deposit & Stake:** `SherpaVault::depositAndStake` pulls USDC from user via `SherpaUSD::depositToVault`, mints SherpaUSD to the vault's keeper, increments vault's `totalPending`, and records a stake receipt for the creditor.
- **Round Roll (Primary chain):** Operator calls `rollToNextRound(yield, globalTotals...)`. The vault:
  - computes new global price
  - sets `roundPricePerShare[currentRound]`,
  - mints shares to the vault for local pending deposits,
  - adjusts vault's SherpaUSD via mint/burn for yield,
  - clears pending and advances `round`.
- **Round Apply (Secondary chains):** Operator calls `applyGlobalPrice(newRound, price)`, which:
  - validates round progression and min-supply,
  - mints shares for local pending at global price,
  - moves pending into `totalStaked`, clears pending, advances `round`.
- **claimShares:** `claimShares(numShares)` or `maxClaimShares()` converts deposit receipts into shares using the price of the round after deposit and moves unredeemed shares from the user's receipt into the user's wallet. Note that the price used to convert to shares is not known to the user at time of depositing, only when

claiming. If the price is unfavorable, the user can however immediately withdraw and receive the original assets back (after waiting for another epoch/round roll).

- **Unstake & Withdraw:**
  - **Two-step (default):** `unstakeAndWithdraw(shares, minOut)` burns shares, transfers SherpaUSD to the wrapper, which creates a withdrawal receipt for the user; user completes next epoch.
  - **Direct (independence on):** `unstake(shares, minOut)` transfers SherpaUSD straight to user.
  - **Instant Unstake (pending only):** `instantUnstake` / `instantUnstakeAndWithdraw` returns pending 1:1 before round conversion.
- **Cross-Chain Rebalancing:**
  - Operator uses wrapper `ownerMint/Burn(vault, amount)` to move SherpaUSD balances across chains (external bridge/pool actually moves tokens).
  - Vault then calls `adjustTotalStaked(amount)` (must match approved amount) and `adjustAccountingSupply(sharesFrom(approvedAmount, lastPrice))` (must match computed shares). Approvals are consumed per vault.

4.4 Centralization Risk

Owner and operator keys control round rolling, pricing, pausing, rebalancing, wrapper transfers, and role changes (operator/keeper). Misuse or compromise can impact user funds and critical functions of the system. A multisig owner, clear on-chain monitoring, and strict operational procedures are recommended.

5 Audit Scope

```
contracts/lib/ShareMath.sol
contracts/lib/Vault.sol
contracts/SherpaUSD.sol
contracts/SherpaVault.sol
```

6 Executive Summary

Over the course of 7 days, the Cyfrin team conducted an audit on the Sherpa smart contracts provided by Sherpa. In this period, a total of 13 issues were found.

During the audit we found three medium-severity findings, centered on owner/operator authority and cross-chain accounting: same-block admin call chaining; unsafe accounting approvals during asset-only rebalances; and a withdrawal flow that effectively requires the primary chain once yield accrues.

In addition, we identified several low-severity issues and informational items.

An additional commit `d9eccb9` was also done during the mitigations. An `owner` parameter was added to the constructor of SherpaVault to allow for a deterministic `CREATE2` deploy. This commit was reviewed and deemed safe.

After the audit the contracts were moved to a public repo: sherpa-vault-smartcontracts-v1.0, the contracts at commit `33dffed` were verified to be the same as the last audit fix commit (`d9eccb9`) reviewed in the private repo.

Summary

Project Name	Sherpa
Repository	sherpa-vault-smartcontracts
Commit	50eb8ad6ee04...
Fix Commit	d9eccb9a9c99...
Audit Timeline	Nov 13th - Nov 21st, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	3
Informational	6
Gas Optimizations	1
Total Issues	13

Summary of Findings

[M-1] Owner can rescue the vault's own share tokens	Resolved
[M-2] Owner can chain admin calls for same-block drains	Resolved
[M-3] Withdrawals can effectively only happen on the primary chain after any yield has accrued	Resolved
[L-1] Misconfigured decimal scale can skew vault accounting	Resolved
[L-2] SherpaUSD does not work with fee-on-transfer tokens	Resolved
[L-3] Direct amount assignment in `SherpaUSD::ownerMint/ownerBurn` can break accounting for totalStaked and accountingSupply	Resolved
[I-1] `SherpaVault::_rollInternal` price calculation comment and math inconsistent	Resolved
[I-2] `SherpaUSD::consumeTotalStakedApproval` and `SherpaUSD::consumeAccountingApproval` callable by anyone	Resolved
[I-3] `CCIPReceiver` dependency not necessary	Resolved
[I-4] `SherpaVault::redeem` naming ambiguous	Resolved
[I-5] Some SherpaUSD can never be unstaked due to minimumSupply check	Resolved
[I-6] Consider implementing explicit rounding behaviour instead of default round down	Resolved
[G-1] Optimize setters by emitting event before state updates	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Owner can rescue the vault's own share tokens

Description: `SherpaVault::rescueTokens` forbids rescuing the wrapper token:

```
// CRITICAL: Cannot rescue the wrapper token (user funds)
// This protects deposited SherpaUSD from being withdrawn by owner
if (token == stableWrapper) revert CannotRescueWrapperToken();
```

But it allows rescuing the vault's own share token (`token == address(vault)`). Sincenewly minted shares for pending deposits are held in vault custody at `address(this)` and user redemptions transfer from this balance:

```
accountingSupply += mintShares;
_mint(address(this), mintShares);
```

The owner can transfer out custody shares via `rescueTokens`, reducing the vault-held pool that backs users' pending/redemption balances.

Impact: An owner (or compromised owner key) can move vault-custodied shares away from `address(this)`, which they then can withdraw for the underlying deposit.

Recommended Mitigation: Disallow rescuing the vault's own share token:

```
- if (token == stableWrapper) revert CannotRescueWrapperToken();
+ if (token == stableWrapper || token == address(this)) revert CannotRescueWrapperToken();
```

Sherpa: Fixed in commit `1a634e0`

Cyfrin: Verified. The vault token is now also prevented from being rescued.

7.1.2 Owner can chain admin calls for same-block drains

Description: The protocol's admin controls let the owner chain privileged calls across the vault and wrapper in a single transaction:

- **Vault path:** Call `SherpaVault::setStableWrapper` to switch which token is protected from rescue. Then immediately call `SherpaVault::rescueTokens` to withdraw any balance of the old wrapper from the vault.
- **Wrapper operator path:** Call `SherpaUSD::setOperator`, then (as operator) use `SherpaUSD::transferAsset` to move USDC out of the wrapper.
- **Wrapper keeper path:** Call `SherpaUSD::setKeeper`, then use `SherpaUSD::depositToVault` to pull USDC from users who left approvals, mint SherpaUSD to the keeper, and extract value via the `transferAsset` path above.

All of these are owner-only and have no built-in delay, so they can be executed together in the same block.

Impact: Even though the code comments stress limiting owner power, the owner (or a compromised key) can immediately redirect custody and move funds with no user warning or reaction time. This creates a trust gap between stated intent and actual authority.

Recommended Mitigation: * Add a delay (at least one withdrawal epoch) to `SherpaVault.setStableWrapper`, `SherpaVault.rescueTokens`, `SherpaUSD.setOperator`, `SherpaUSD.setKeeper`, and consider delaying `SherpaUSD.transferAsset`.

- Make `SherpaVault.stableWrapper`, `SherpaUSD.keeper` immutable.
- Use a timelock (e.g., OpenZeppelin `TimelockController`) with a user-protective delay so people can withdraw or reduce approvals before changes take effect.

Sherpa:

> Vault path: Call SherpaVault::setStableWrapper to switch which token is protected from rescue. Then immediately call SherpaVault::rescueTokens to withdraw any balance of the old wrapper from the vault. Wrapper keeper path: Call SherpaUSD::setKeeper, then use SherpaUSD::depositToVault to pull USDC from users who left approvals, mint SherpaUSD to the keeper, and extract value via the transferAsset path above.

We're implementing a pseudo-immutable `stableWrapper` and `keeper` - both will be set once during deployment and cannot be changed after system initialization. This eliminates both attack surfaces while maintaining the deployment flexibility needed to solve the chicken-and-egg deployment problem: vault constructor requires wrapper address, but we can't deploy wrapper until vault exists. We solve this by deploying vault with a temporary wrapper address, then calling `setStableWrapper()` once to set the real wrapper and lock it permanently.

> Wrapper operator path: Call SherpaUSD::setOperator, then (as operator) use SherpaUSD::transferAsset to move USDC out of the wrapper.

Timelocks / delays on `setOperator` and related admin functions would be ineffective given our vault's trust model and architecture. The operator already has manual custody of strategy funds (transferred to fund manager for on and off-chain strategy delegation) and can pause the system at will, meaning any timelock delay could be circumvented by simply pausing withdrawals during the timelock window. The operator must remain changeable for operational flexibility (personnel changes, key rotation) so we cant make it immutable like we did with `keeper` and `setStableWrapper`. The owner role is a 2-of-3 multisig that controls operator selection, so centralization is lessened there as best as we can.

Fixed in commit `15e2706`

Cyfrin: Verified. Both `stableWrapper` and `keeper` now locked after initial assignment which will effectively make them immutable. Operator concern acknowledged.

7.1.3 Withdrawals can effectively only happen on the primary chain after any yield has accrued

Description: During round rolls, yield is only realized on the primary chain in `SherpaVault::_adjustBalanceAndEmit`. This leaves the system in a problematic state if withdrawals happens on another chain.

Imagine the scenario: there's 500 + 500 deposits of SherpaUSD on chain A and B, A being primary. 100 SherpaUSD is added as yield on A. The balance is 500 + 600, global total 1100 giving a share price of 1.1. Alice, who has half the total shares decides do withdraw on chain B, giving her 550 SherpaUSD (USDC). Since this isn't available on chain B, the protocol needs to rebalance 50 SherpaUSD from A to B.

They do this by calling `SherpaUSD::ownerBurn(50)` on chain A followed by `SherpaUSD::ownerMint(50)` on chain B. This will store 50 in both `approvedTotalStakedAdjustment` and `approvedAccountingAdjustment` on both chains. The latter one being the issue.

Once `SherpaVault::adjustTotalStaked` is called by the operator, the rebalance of SherpaUSD is done, and Alice can effectively withdraw. However, there's no way to clear the state in `approvedAccountingAdjustment` as no shares were ever moved. If `SherpaVault::adjustAccountingSupply` is called, it will corrupt the `accountingSupply` as no shares were ever moved. So the states of `approvedAccountingAdjustment` are effectively permanently corrupted as `consumeAccountingApproval` can only be cleared from the vault.

In addition to this, if `SherpaVault::adjustAccountingSupply` was called on chain A, `accountingSupply` would be decremented and the `accountingSupply` subtraction in function `_unstake()` would underflow on chain A, hence bricking funds.

Impact: Withdrawals can only safely happen on the primary chain as soon as any yield is accrued. If yield is withdraw from the secondary chain that will corrupt either `SherpaUSD.approvedAccountingAdjustment` or `SherpaVault.accountingSupply` on both chains.

Recommended Mitigation: Consider split approval modes. Introduce explicit asset-only rebalancing (set `approvedTotalStakedAdjustment` without setting `approvedAccountingAdjustment`) and a share-sync mode (set both).

Sherpa: Fixed in commit `34f2092`

Cyfrin: Verified. Calls to rebalance assets only were added.

7.2 Low Risk

7.2.1 Misconfigured decimal scale can skew vault accounting

Description: The vault's math assumes the same decimal scale as the wrapped asset (USDC, 6 decimals) and as the `globalPricePerShare` fed by ops. While deployment sets `vaultParams.decimals = 6` and the wrapper enforces USDC's 6 decimals, a misconfiguration will skew conversions.

Impact: Configuring the vault with more than 6 decimals can cause incorrect accounting, and follow-on reverts in rebalancing.

Recommended Mitigation: Consider locking the vault decimals to 6, same as `SherpaUSD`.

Sherpa: Fixed in commit 1a634e0

Cyfrin: Verified. `_vaultParams.decimals` now verified to be 6 in the constructor.

7.2.2 SherpaUSD does not work with fee-on-transfer tokens

Description: The SherpaUSD contract cannot work correctly with fee-on-transfer tokens. An example of such a token is USDT, which is expected to be supported as per comments. Note: Fees are not yet activated on USDT however they can be at any time in the future.

```
        // CRITICAL: SherpaUSD only supports 6-decimal assets (USDC, USDT, etc.)
```

For example:

- Assume 2% fees are charged by a fee-on-transfer token.
- Keeper calls function depositToVault with 100e6 as amount.
- 100 SherpaUSD are minted to the keeper
- Due to fees charged on transfer, only 98 tokens are received by the contract.
- This can build up over time and cause late withdrawers to incur a loss as they will be unable to withdraw fully or a partial amount of their tokens.

```
function depositToVault(
    address from,
    uint256 amount
) external nonReentrant onlyKeeper {
    if (amount == 0) revert AmountMustBeGreaterThanZero();

    _mint(keeper, amount);
    depositAmountForEpoch += amount;

    emit DepositToVault(from, amount);

    IERC20(asset).safeTransferFrom(from, address(this), amount);
}
```

Recommended Mitigation: Consider adding support for fee-on-transfer tokens. Alternatively consider not supporting such tokens.

Sherpa: Fixed on commit 0b32641

Cyfrin: Verified. Comment changed to explicitly say FOT tokens not supported (including USDT).

7.2.3 Direct amount assignment in `SherpaUSD::ownerMint/ownerBurn` can break accounting for totalStaked and accountingSupply

Description: Functions `SherpaUSD::ownerMint` and `ownerBurn` directly assign the amount parameter to mappings `approvedTotalStakedAdjustment` and `approvedAccountingAdjustment`. This will however not work correctly if more tokens are minted or burned to/from the vault before the approvals are consumed.

For example:

- Operator mints 100 SherpaUSD to SherpaVault.
- This tracks `approvedTotalStakedAdjustment` and `approvedAccountingAdjustment` as 100 SherpaUSD each
- Operator performs another mint of 200 tokens before the previous approvals are consumed.
- Now the issue is that approvedTotalStakedAdjustment and approvedAccountingAdjustment will be overwritten to store 200 SherpaUSD each instead of 300 SherpaUSD.
- This is clearly incorrect and breaks accounting since old approvals were not consumed yet by the vault.

```
function ownerMint(address to, uint256 amount) external onlyOperator {
    _mint(to, amount);

    // Approve vault to adjust by this amount
    approvedTotalStakedAdjustment[to] = amount;
    approvedAccountingAdjustment[to] = amount;

    emit PermissionedMint(to, amount);
    emit RebalanceApprovalSet(to, amount, amount);
}

/**
 * @notice Operator-level burn for manual rebalancing across chains
 * @param from Address to burn from
 * @param amount Amount to burn
 * @dev Sets approval for vault to adjust totalStaked and accountingSupply
 */
function ownerBurn(address from, uint256 amount) external onlyOperator {
    _burn(from, amount);

    // Approve vault to adjust by this amount
    approvedTotalStakedAdjustment[from] = amount;
    approvedAccountingAdjustment[from] = amount;

    emit PermissionedBurn(from, amount);
    emit RebalanceApprovalSet(from, amount, amount);
}
```

Recommended Mitigation: Consider replacing direct amount assignments with the += and -= operators in ownerMint and ownerBurn respectively.

Sherpa: Fixed in commit `1cd0018`

Cyfrin: Verified. Checks to enforce that the approvals have been consumed are added. This prevents any accounting corruption.

7.3 Informational

7.3.1 `SherpaVault::_rollInternal` price calculation comment and math inconsistent

Description: When calculating a new price a script queries all vaults on all chains then passes that to `SherpaVault:: rollToNextRound`. This in turn calls `SherpaVault::_rollInternal` where the new price is calculated:

```
// Calculate global price using script-provided totals
// globalBalance must include pending deposits for correct price calculation
uint256 globalBalance = isYieldPositive
    ? globalTotalStaked + globalTotalPending + yield
    : globalTotalStaked + globalTotalPending - yield;

uint256 newPricePerShare = ShareMath.pricePerShare(
    globalShareSupply,
    globalBalance,
    globalTotalPending,
    _vaultParams.decimals
);
```

The code comments state that `globalBalance must include pending deposits` yet `globalBalance` is passed to `ShareMath:pricePerShare`, which immediately subtracts the `pending` amount: `((totalBalance - pending) / totalSupply)`:

```
function pricePerShare(
    uint256 totalSupply,   // @audit-info globalShareSupply
    uint256 totalBalance,  // @audit-info globalBalance
    uint256 pendingAmount, // @audit-info globalTotalPending
    uint256 decimals
) internal pure returns (uint256) {
    uint256 singleShare = 10 ** decimals;
    return
        totalSupply > 0
            ? (singleShare * (totalBalance - pendingAmount)) / totalSupply
            : singleShare;
}
```

The comment in `_rollInternal` is inconsistent with the math applied as the actual price calculation doesn't include the `pendingAmount`.

Consider changing the comment or if the comment is correct, the math.

Sherpa: Fixed in commit `9dbaf27`

Cyfrin: Verified. Comment was incorrect and is not fixed.

7.3.2 `SherpaUSD::consumeTotalStakedApproval` and `SherpaUSD::consumeAccountingApproval` callable by anyone

Description: In the rebalancing/settlement flow for cross-chain accounting, `SherpaUSD::consumeTotalStakedApproval` and `SherpaUSD::consumeAccountingApproval` serve as the "consume/clear" step for one-time approvals set by `SherpaUSD::ownerMint/SherpaUSD::ownerBurn`. They are invoked around `SherpaVault::adjustTotalStaked` and `SherpaVault::adjustAccountingSupply` to prevent reuse of an approval after the corresponding adjustment is applied.

Both functions are externally callable and accept a `vault` parameter, but state changes are gated by `if (msg.sender != vault) revert OnlyVaultCanConsume();` and approvals are keyed by the caller address:

```
function consumeTotalStakedApproval(address vault) external {
    // @audit-issue anyone can call by passing their own address as `vault`
    if (msg.sender != vault) revert OnlyVaultCanConsume();
    approvedTotalStakedAdjustment[vault] = 0;
    emit TotalStakedApprovalConsumed(vault);
}
```

Consequently, any address may call the functions, yet the call can only clear its own approval entry, not a vault's. Behavior is correct and non-exploitable in this design; however, the open callable surface combined with an explicit `vault` parameter can be confusing to integrators and reviewers.

Consider removing the `vault` parameter and only allow the actual vault to call by adding the `onlyKeeper` modifier. This would follow the principle of least privilege and limit the attack surfaces available.

Sherpa: Fixed in commit c33eb52

Cyfrin: Verified. Both functions now have the `vault` parameter removed and the `onlyKeeper` modifier.

7.3.3 `CCIPReceiver` dependency not necessary

Description: `SherpaVault` inherits `CCIPReceiver`, but the protocol's cross-chain flow uses CCIP burn/mint token pools rather than ad-hoc message passing. Chainlink's cross-chain token pattern on EVM chains does not require a `CCIPReceiver` implementation on the token/vault contract, only pool authorization via `mint/burn` style hooks. Keeping `CCIPReceiver` (and its `_ccipReceive` stub) increases bytecode size, deployment cost, and surface area without delivering any functionality.

Consider removing the inheritance and associated code to simplify the contract, reduce gas/bytecode footprint, and avoid implying a message-bridge dependency that isn't actually used.

Sherpa: Removed in commit 59974b2

Cyfrin: Verified. `CCIPReceiver` dependency now removed.

7.3.4 `SherpaVault::redeem` naming ambiguous

Description: `SherpaVault` uses ERC-4626-adjacent terminology but different semantics. In ERC-4626, `redeem` means burning shares to withdraw assets. In `SherpaVault`, `redeem` means finalize a prior deposit by moving unredeemed shares into the user's wallet. This naming can mislead integrators and tooling that assume ERC-4626 behavior.

Consider renaming `redeem` to `finalizeDeposit` / `claimShares` to prevent confusion.

Sherpa: Fixed in commit 8e9ba92

Cyfrin: Verified. `claimShares` now used.

7.3.5 Some SherpaUSD can never be unstaked due to minimumSupply check

Description: The `SherpaVault::_unstake` function in SherpaVault includes a check that ensures the total assets staked are never less than minimumSupply and greater than 0. However, it is possible for another user to intentionally or unintentionally block a user from unstaking permanently.

```
// Ensure vault maintains minimum supply (allow full exit to 0)
        if (totalStaked - wrappedTokensToWithdraw < vaultParams.minimumSupply &&
            totalStaked - wrappedTokensToWithdraw > 0) {
            revert MinimumSupplyNotMet();
        }
```

For example:

- Let's assume `minimumSupply` = 1000 SherpaUSD.
- Alice deposits 1000 SherpaUSD.
- Malicious Bob deposits 1 wei SherpaUSD. This is allowed since this if statement in function `_stakeInternal` - `if (totalWithStakedAmount < _vaultParams.minimumSupply) revert MinimumSupplyNotMet();` checks the total staked supply + pending amount i.e. the totalWithStakedAmount, which is now 1000 SherpaUSD + 1 wei SherpaUSD.

- Alice now cannot exit the system until Bob clears his withdrawal. This occurs due to the minimumSupply check in the `_unstake` function.
- Alice can only withdraw 1 wei SherpaUSD while the remaining is permanently locked.

Based on the scripts shared, this issue does not pose a risk currently as `minimumSupply` is expected to be 1 USD.

Recommended Mitigation: It is recommended to implement either or both of the following recommendations as a safety measure:

1. Implement a setter function to keep the `minimumSupply` configurable.
2. Add check to ensure all users individually deposit above the minimum supply.

Sherpa: Fixed in commit 720c2c0

Cyfrin: Verified. A minimum deposit enforced. `minimumSupply` left immutable.

7.3.6 Consider implementing explicit rounding behaviour instead of default round down

Description: All functions in `ShareMath.sol` round down to the nearest integer currently. This can be unfavourable in certain instances for the SherpaVault.

For example, the `ShareMath.pricePerShare` function uses integer division which causes precision loss. Hence it would slightly underestimate the price per share.

```
function pricePerShare(
    uint256 totalSupply,
    uint256 totalBalance,
    uint256 pendingAmount,
    uint256 decimals
) internal pure returns (uint256) {
    uint256 singleShare = 10 ** decimals;
    return
        totalSupply > 0
            ? (singleShare * (totalBalance - pendingAmount)) / totalSupply
            : singleShare;
}
```

Recommended Mitigation: It is recommend to perform explicit rounding in addition to adding comments that logically elaborate why the respective rounding direction is appropriate in each instance.

Sherpa: Fixed in commit 61345a1

Cyfrin: Verified. Documentation about the specific rounding directions added.

7.4 Gas Optimization

7.4.1 Optimize setters by emitting event before state updates

Description: Functions `SherpaUSD::setKeeper`, `SherpaUSD::setOperator`, `SherpaUSD::setAutoTransfer` and `SherpaVault::setDepositsEnabled`, `SherpaVault::setStableWrapper` create an unnecessary memory variable to store old values used for event emissions. However, this is not required if the event is emitted first.

For example, function setKeeper can be optimized in the following manner:

```
function setKeeper(address _keeper) external onlyOwner {
    if (_keeper == address(0)) revert AddressMustBeNonZero();
    emit KeeperSet(keeper, _keeper);
    keeper = _keeper;
}
```

Recommended Mitigation: Consider removing the memory variables by emitting events first.

Sherpa: Fixed in commit 7e34a6b

Cyfrin: Verified.