Button Basis Trade Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

ChainDefenders (0x539 & PeterSR)

September 25, 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

4.1 Protocol summary

BasisTrade is a HyperEVM-integrated asset router built around an ERC-4626 vault. Users deposit a base asset (e.g., USDC) into `BasisTradeVault`, which mints shares (with optional deposit/withdrawal fees), enforces an optional deposit whitelist and TVL cap, and tracks withdrawal requests via a queue. Funds intended for trading are pushed from the vault to a per-vault `Pocket` wallet through `BasisTradeTailor`. The Tailor then interacts with HyperLiquid HyperCore (encoded by `CoreWriterEncoder`) to: add API wallets, move tokens to/from Core spot wallet, and transfer USD class balances between spot and perp. Withdrawals from Vault are two-step: users request redemption (shares are escrowed and assets "promised"), agents refill the vault from Core via the Tailor and then process the queue to pay users from the vault.

4.2 Actors and Roles

- **1. Actors:**
  - **BasisTrade team:** Deploys upgradeable contracts, configures parameters, manages roles, and approves Tailors in the PocketFactory.
  - **Depositors (vault users):** Deposit the base asset to receive ERC-4626 shares and request withdrawals via the queue.
  - **Agents (operators):** Execute operational flows: funding the Pocket, moving funds to/from Core, adding API wallets, and processing the withdrawal queue.
- **2. Roles (contract-level):**
  - **Owner / DEFAULT_ADMIN_ROLE (Vault & Tailor):** Can upgrade (UUPS), set fees/TVL cap/oracle (Vault), manage creation whitelist (Tailor), and grant/revoke subordinate roles.
  - **ADMIN_ROLE:** Manages configuration and oracle, can grant/revoke **AGENT_ROLE**.
  - **AGENT_ROLE:** Performs live operations (Core actions via Tailor, vault funding, queue processing).

4.3 Key Components

- **BasisTradeVault (ERC-4626, UUPS):** Manages deposits/fees, optional deposit whitelist + TVL cap, withdrawal queue with share escrow, and oracle-based AUM (`totalAssets`). Provides admin hooks to set fees, oracle, caps; agent hooks to deposit to Tailor/Pocket and update withdrawal intent mirrored in Tailor. Creates its dedicated Pocket via the Tailor. The vault is not fully ERC-4626 compliant—`withdraw`, `redeem`, and `mint` are disabled (and `previewMint` currently reverts), which some ERC-4626 tooling expects. Additionally, `totalAssets()` can revert (due to missconfiuration or oracle issues), so integrators should handle this behavior.
- **BasisTradeTailor (UUPS):** Orchestrates Pocket creation (via `PocketFactory`), maps each Pocket to a "user" (the Vault in this setup), and executes HyperCore actions by calling the CoreWriter through the Pocket (`pocket::exec`). Supports: adding API wallets, spot token sends to/from Core, and USD class transfers between spot/perp. Tracks an aggregate withdrawal intent per Pocket.
- **Pocket & PocketFactory:** Pocket is a minimal, ownable smart-wallet (clones) that can `transfer`, `approve`, and `exec` arbitrary calls (no ETH). `PocketFactory` restricts Pocket creation to approved Tailors and records factory-created Pockets.

4.4 Centralization risk

Core control is concentrated in upgradeable admin keys: owners/default admins can upgrade implementations; admins set fees, TVL caps, and the oracle that defines AUM; and agents directly move funds between the vault, Pocket, and HyperCore and process withdrawals. Operational safety therefore depends on multisig/timelock hygiene, strict key management, and monitoring. The system also relies on the configured oracle to reflect Pocket/Core balances accurately; stale/misconfigured oracles can distort AUM and share accounting.

5 Audit Scope

```
src/libraries/CoreWriterEncoder.sol
src/BaseTailor.sol
src/BasisTradeTailor.sol
src/BasisTradeVault.sol
src/Pocket.sol
src/PocketFactory.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Button Basis Trade smart contracts provided by Button. In this period, a total of 20 issues were found.

During the audit we identified 1 high-severity issue, 1 medium-severity design risk, and several low and informational items. The high finding is a head-of-line blocking risk in the withdrawal queue: a single reverting transfer can permanently stall processing for everyone behind it.

The medium item concerns governance: mixing Ownable with AccessControl creates dual gatekeeping for privileged actions, which is brittle and increases the chance of lock-outs or misconfiguration.

The low-severity issues, in brief: withdrawals priced at execution can overpay during sharp drawdowns; missing minimum-deposit guard leaves room for dust/rounding abuse; ERC-165 reporting in Tailor is incomplete; and the Vault-Tailor withdrawal counters can drift unless the design is made delta-based or single-sourced and safer key handling. Informational items are mostly hardening, interoperability and code quality.

The Cyfrin team also pushed an implementation of `mint` together with two fuzz tests verifying the rounding directions of the vault in PR#1.

Summary

Project Name	Button Basis Trade
Repository	button-protocol
Commit	9002f2b0d05b...
Fix Commit	a349cf2c58fe...
Audit Timeline	Sep 15th - Sep 19th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	1
Low Risk	7
Informational	10
Gas Optimizations	1
Total Issues	20

Summary of Findings

[H-1] Single reverting withdrawal can block the `BasisTradeVault` withdrawal queue	Resolved
[M-1] Combination of Ownable and AccessControl can cause loss of admin functionality	Resolved
[L-1] Missing minimum deposit enforcement	Resolved
[L-2] `BasisTradeTailor` is ERC-165 non compliant	Resolved
[L-3] State drift between `BasisTradeVault.totalPendingWithdrawals` and `BasisTradeTailor.withdrawalRequests[pocket]`	Resolved
[L-4] Decimal mismatch for tokens on HyperEVM and HyperCore	Acknowledged
[L-5] Deployment script requires unencrypted private keys	Resolved
[L-6] Withdrawals priced at execution problematic during large price swings	Resolved
[L-7] Lack of check for 0 shares minted	Resolved
[I-1] Redundant variable statements	Resolved
[I-2] Missing cancellation of withdrawal and redeem requests	Acknowledged
[I-3] Consider using exponential notation in tests	Resolved
[I-4] Unused `Pocket::approve` function	Acknowledged
[I-5] Consider implementing `ERC4626::mint`	Resolved

[I-6] Consider enabling on-behalf-of withdrawals	Resolved
[I-7] Missing SPDX License Identifiers	Acknowledged
[I-8] `BasisTradeVault::totalAssets` may undercount pocket-held funds	Resolved
[I-9] Use named mapping parameters to explicitly note the purpose of keys and values	Resolved
[I-10] Remove obsolete return statements when using named return variables	Resolved
[G-1] Redundant `approve(0)` in `BasisTradeVault::depositToTailor`	Resolved

7 Findings

7.1 High Risk

7.1.1 Single reverting withdrawal can block the `BasisTradeVault` withdrawal queue

Description: `BasisTradeVault::processWithdrawal` processes exactly one request at the queue head and performs the final ERC20 `safeTransfer` to the request's `user`. If that transfer reverts, the whole tx reverts and the head entry remains in place. Because the function always targets `queueHead` and provides no way to skip, quarantine, or edit the failing entry, a single reverting withdrawal permanently blocks the entire queue (head-of-line blocking). Common revert causes include:

- The receiver is blacklisted/blocked by the token (e.g., USDC/USDT compliance lists).
- The computed `assets` for a request becomes `0` due to rounding/fees, and the token reverts on zero-amount transfers.

This can happen accidentally or be used to grief the protocol by placing an unprocessable request at the head. The queue remains stuck until a contract upgrade or manual intervention.

Impact: Withdrawal processing can be indefinitely halted for all users behind the stuck request, causing severe withdrawal delays and potential loss of user confidence.

Recommended Mitigation: Consider implementing the following:

- Redesign away from a strict queue to a timelock + user-pull/admin-push model: Record unlockable claims and let each user call `processWithdrawal` themselves after the timelock.
- Add a skip/quarantine mechanism: if a head withdrawal fails, move it into a "frozen" set (keeping shares escrowed and assets reserved), advance `queueHead`, and allow others to proceed. Provide functions for the user to update their payout address and for agents to retry/cancel within policy.

Button: Fixed in commit `9cde24c` by moving to a request based system. Did not add the ability for users to manually claim withdrawals. If this is something that becomes needed, we can upgrade the contract to support it fairly easily with the request pattern we have.

Cyfrin: Verified. The queue is now removed and the withdrawals are done on a per-request basis by the agent.

7.2 Medium Risk

7.2.1 Combination of Ownable and AccessControl can cause loss of admin functionality

Description: `BasisTradeTailor` and `BasisTradeVault` mix `Ownable(2Step)Upgradeable` with `AccessControlUpgradeable`. Several admin wrappers are `onlyOwner` but internally call `grantRole` / `revokeRole`, which themselves require the caller to hold the role's admin (usually `DEFAULT_ADMIN_ROLE`). Example:

```
function grantAdmin(address account) external onlyOwner {
    grantRole(ADMIN_ROLE, account); // requires DEFAULT_ADMIN_ROLE too
}
```

This creates a dual requirement: the caller must be both `owner` and `DEFAULT_ADMIN_ROLE`. If those identities diverge (e.g., ownership transferred without also granting default admin), governance gets brittle and confusing.

Impact: The "Owner" may be unable to manage roles (grant/revoke admin/agent) and a `DEFAULT_ADMIN_ROLE` holder who isn't `owner` can't upgrade (since `_authorizeUpgrade` is `onlyOwner`) as well as extra wrappers duplicate functionality and enlarge the attack surface/bytecode/ABI for no gain.

Proof of Concept: Add the following test to `BasisTradeVault.t.sol` (a very similar test would work for Tailor):

```
function test_OwnerLosesGrantAbility() public {
    _setupComplete();

    // owner transfers ownership
    vm.prank(deployer);
    vault.transferOwnership(alice);
    assertEq(vault.owner(), alice);

    // new owner cannot grant admin (or agent) roles
    vm.prank(alice);
    vm.expectRevert(
        abi.encodeWithSelector(
            IAccessControl.AccessControlUnauthorizedAccount.selector,
            alice,
            bytes32(0x00)
        )
    );
    vault.grantAdmin(bob);
}
```

Recommended Mitigation: Consider unifying on AccessControl by removing Ownable entirely. Gate privileged functions with `onlyRole(DEFAULT_ADMIN_ROLE)` and authorize upgrades via the same role:

```
function grantAdmin(address account) external onlyRole(DEFAULT_ADMIN_ROLE) {
  grantRole(ADMIN_ROLE, account);
}

function _authorizeUpgrade(address impl) internal override onlyRole(DEFAULT_ADMIN_ROLE) {}
```

Then delete the bespoke wrappers `grantAdmin`, `revokeAdmin`, `grantAgent`, `revokeAgent` altogether. `DEFAULT_ADMIN_ROLE` already has authority to call `grantRole` / `revokeRole` directly, so these wrappers are redundant. Removing them shrinks bytecode/ABI, reduces surface area, and tidies the contracts. To prevent lockout: use `AccessControlEnumerableUpgradeable` and enforce at least one default admin remains:

```
function _revokeRole(bytes32 role, address account) internal override {
    if (role == DEFAULT_ADMIN_ROLE) {
        require(getRoleMemberCount(DEFAULT_ADMIN_ROLE) > 1, "keep >=1 default admin");
    }
    super._revokeRole(role, account);
}
```

Or if keeping `Ownable` is prefered, synchronize roles on ownership changes (grant new owner `DEFAULT_ADMIN_ROLE` and revoke from old).

Button: Fixed in commit `32f8ca9` by moving to AccessControlEnumerableUpgradeable only.

Cyfrin: Verified. `AccessControlEnumerableUpgradeable` now used for both Tailor and Vault. The grant/revoke calls also removed in favor or AccessControls own calls.

7.3 Low Risk

7.3.1 Missing minimum deposit enforcement

Description: The `BasisTradeVault::deposit` function does not enforce a minimum deposit amount. Allowing dust deposits can lead to several undesirable situations:

1. **Economic Unviability:** A user might deposit an amount so small that the gas fees for the transaction are significantly higher than the value of the deposit itself.
2. **Potential for Nuisance:** It could enable scenarios where an attacker spams the vault with many tiny deposits, which, while not a direct security threat, can be a nuisance.

Although the contract is protected from the most severe issues by the base ERC4626 implementation, enforcing a sensible minimum deposit amount is a good practice for user protection and contract robustness.

Recommended Mitigation: Introduce a new state variable, `minDepositAmount`, which can be set by an admin. Modify the `deposit` function to require that the deposited `assets` are greater than or equal to this minimum amount.

Button: Fixed in commit `9cde24c`.

Cyfrin: Verified. A minimum deposit configurable by admin is now enforced

7.3.2 `BasisTradeTailor` is ERC-165 non compliant

Description: The `BasisTradeTailor` contract inherits from and implements the `ITailor` interface. According to the ERC-165 standard, the `supportsInterface` function should return `true` when queried with the interface ID of `ITailor` and `IERC1822Proxiable` (coming from `UUPSUpgradeable`).

However, the current implementation of `supportsInterface` only calls `super.supportsInterface(interfaceId)`, which delegates the check to the parent `AccessControlUpgradeable` contract. The parent contract is unaware of the `ITailor` and `IERC1822Proxiable` interfaces and will therefore return `false` for the interface IDs. This means the contract incorrectly reports that it does not support interfaces it actually implements, which can break interactions with other contracts that rely on ERC-165 for interface detection.

Recommended Mitigation: The `supportsInterface` function should be updated to explicitly check for the `ITailor` and `IERC1822Proxiable` interface IDs in addition to calling the `super` function. This ensures that the contract correctly advertises its implementation of the given interfaces.

```
// ...existing code...

import {IERC1822Proxiable} from "@openzeppelin/contracts/interfaces/draft-IERC1822.sol";

// ...existing code...

    /**
     * @notice Override supportsInterface to resolve multiple inheritance
     */
    function supportsInterface(bytes4 interfaceId)
        public
        view
        override(AccessControlUpgradeable)
        returns (bool)
    {
        return interfaceId == type(ITailor).interfaceId ||
        return interfaceId == type(IERC1822Proxiable).interfaceId ||
        super.supportsInterface(interfaceId);
    }

// ...existing code...
```

Button: Fixed in commit `32f8ca9`

Cyfrin: Verified. Recommendation implemented.

7.3.3 State drift between `BasisTradeVault.totalPendingWithdrawals` and `BasisTradeTailor.withdrawalRequests[pocket]`

Description: `BasisTradeVault` tracks pending withdrawals in `totalPendingWithdrawals` while `BasisTradeTailor` keeps its own `withdrawalRequests[pocket]`. These two counters are mutated on different code paths:

- `BasisTradeVault::requestRedeem` bumps the Vault counter and sets the Tailor counter to a new total.
- `BasisTradeVault::processWithdrawal` only reduces the Vault counter.
- `BasisTradeTailor::processWithdrawal` only reduces the Tailor counter.

Although the Vault exposes `updateWithdrawalRequest(uint256 amount)`, it still uses a set-the-total model, which is race-prone and allows the two aggregates to drift with normal operations. Manual re-syncs with "set total" are brittle and can themselves overwrite correct values during concurrent requests.

Consider one of the following redesigns:

- Make the Vault the single source of truth and remove Tailor's `withdrawalRequests` entirely. Tailor acts only as an executor to move funds; bots and operators read only `vault::totalPendingWithdrawals`.
- Switch from set total to delta-based accounting guarded by the Vault: Have `tailor::processWithdrawal` increase by a delta instead, and have the vault send the delta from `requestRedeem`.

Button: Fixed in ". Updated to have only the offchain agent set the withdrawal request amount. Will need this to be able to do net settling between deposits and withdrawals.

Cyfrin: Verified. `totalPendingWithdrawals` removed in BasisTradeVault'.

7.3.4 Decimal mismatch for tokens on HyperEVM and HyperCore

Description: The `transferToCore` function transfers a specified `amount` of the base asset from a pocket to HyperCore. However, it does not account for potential differences in decimal precision between the HyperEVM token and the HyperCore token. If the two systems use different decimal configurations (e.g., 6 decimals vs. 18 decimals), the transferred `amount` may represent a drastically different value on HyperCore than intended on HyperEVM. This can result in either loss of funds or inflation of balances, depending on the mismatch.

Impact: Agents could unintentionally transfer more or fewer tokens than expected due to mismatched decimals.

Recommended Mitigation: Consider introducing a decimal normalization mechanism when transferring between HyperEVM and HyperCore.

Button: Acknowledged. As there is not much on-chain info available for the HyperCore decimals we will solve this off-chain in the agent.

7.3.5 Deployment script requires unencrypted private keys

Description: Several deployment/ops scripts require private keys to be loaded from environment variables and used directly inside the script, e.g.:

```
// DeployBasisTradeTailor.s.sol
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
uint256 adminPrivateKey    = vm.envUint("ADMIN_PRIVATE_KEY");
...
vm.startBroadcast(deployerPrivateKey);
...
vm.startBroadcast(adminPrivateKey);

// DeployBasisTradeVault.s.sol
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
...
vm.startBroadcast(deployerPrivateKey);

// DeployMockPocketOracle.s.sol
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
```

```
...
vm.startBroadcast(deployerPrivateKey);

// DeployMocks.s.sol
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
...
vm.startBroadcast(deployerPrivateKey);
```

Storing and loading raw private keys via `.env` (plain text) is an operational security risk: keys can be leaked through version control, logs, shell history, misconfigured backups, or compromised developer machines/CI runners.

A safer approach is to avoid embedding keys in scripts and use Foundry's wallet management and keystore support. Recommended pattern:

1. Import keys into an encrypted local keystore (once per machine) using `cast`:

```
cast wallet import deployerKey --interactive
cast wallet import adminKey --interactive
cast wallet import agentKey --interactive   # if needed
```

2. Change scripts to use parameterless broadcasting so the signer is supplied by CLI:

```
// before: vm.startBroadcast(deployerPrivateKey);
vm.startBroadcast();
// ...
vm.stopBroadcast();
```

3. Run each role-sensitive phase as the appropriate account (split into separate runs or separate scripts if different signers are required):

```
# Deployer phase
forge script script/DeployBasisTradeTailor.s.sol:DeployBasisTradeTailor \
  --rpc-url "$RPC_URL" --broadcast --account deployerKey --sender <deployer_addr> -vvv

# Admin phase (grants/approvals)
forge script script/ConfigureBasisTradeTailor.s.sol:ConfigureBasisTradeTailor \
  --rpc-url "$RPC_URL" --broadcast --account adminKey --sender <admin_addr> -vvv
```

This keeps private keys encrypted at rest and never exposes them via plaintext environment variables. As alternatives, consider hardware wallets (`--ledger`), and ensure `.env` never contains raw keys in shared environments.

For additional guidance, see this explanation video by Patrick.

Button: Fixed in commit `c89bce0`

Cyfrin: Verified. Keystores are now used for the keys.

7.3.6 Withdrawals priced at execution problematic during large price swings

Description: Withdrawals are "price-locked" at request time: `requestRedeem` stores `shares` and the computed `assetsAfterFee = previewRedeem(shares)` using the at-request exchange rate. When an agent later calls `processWithdrawal`, the vault burns the escrowed `shares` but pays out the stored asset amount, not what those shares are worth at execution.

Impact: If the share price has fallen in the interim (e.g., oracle update, Core PnL loss, depeg), early requesters are effectively overpaid relative to the current price, with the shortfall socialized to remaining shareholders. In extreme drawdowns this can accelerate bank-run dynamics and drain the vault faster than intended possibly to the point of insolvency.

Recommended Mitigation: Consider using price at execution. Store only `shares` at request time and compute `assetsAfterFee` at processing using the current exchange rate (i.e., `previewRedeem(shares)` then). Possibly with execution guardrails with acceptable slippage bounds (protocol default and/or user-provided).

Button: Fixed in commit 9cde24c by moving to pricing at execution.

Cyfrin: Verified. Price now taken at execution.

7.3.7 Lack of check for 0 shares minted

Description: `previewDeposit(assets)` can legitimately return `0` shares for tiny deposits due to rounding and/or deposit fees. If the deposit path doesn't guard against this, a user could transfer assets to the vault and receive 0 shares (an unintended "donation").

Consider adding a check for 0 shares:

```
function previewDeposit(uint256 assets) public view virtual override returns (uint256) {
    uint256 fee = _extractFeeFromTotal(assets, depositFeeBps);
    require(fee < assets, "Deposit fee exceeds assets");
    uint256 shares = super.previewDeposit(assets - fee);
    require(shares > 0, "0 shares");
    return shares;
}
```

Button: Fixed in commit 9cde24c

Cyfrin: Verified. `previewDeposit` now checks that `> 0 shares` are minted.

7.4 Informational

7.4.1 Redundant variable statements

Description: In `BasisTradeVault` the functions `maxMint`, `mint`, `withdraw`, and `redeem` are overrides of the standard ERC4626 interface. In this contract, these functions are intentionally disabled to enforce a custom deposit and withdrawal flow (e.g., using `requestWithdraw` and `requestRedeem` instead of the standard `withdraw` and `redeem`).

Because these functions are disabled and immediately revert or return a fixed value, their parameters (`receiver`, `shares`, `assets`, `owner`) are not used within the function bodies. The code explicitly acknowledges this by placing the parameter names on their own lines (e.g., `receiver;`), which silences compiler warnings about unused variables but is redundant.

Recommended Mitigation: An alternative way to denote unused parameters, which can improve clarity, is to write the function declarations without the parameters (e.g., `foo(uint256, address)`). This is a common convention in Solidity to signal that a parameter is intentionally unused.

```
// ...existing code...
    /**
     * @notice Mint function is disabled
     * @dev This vault only supports asset-based deposits
     */
    function mint(uint256 /*shares*/, address /*receiver*/) public virtual override returns (uint256) {
        revert("Mint disabled: use deposit");
    }

    // ========================================
// ...existing code...
    /**
     * @notice Standard withdraw function is disabled
     * @dev Users must use requestWithdraw instead
     */
    function withdraw(
        uint256 /*assets*/,
        address /*receiver*/,
        address /*owner*/
    ) public virtual override returns (uint256) {
        revert("Withdraw disabled: use requestWithdraw");
    }

    /**
     * @notice Standard redeem function is disabled
     * @dev Users must use requestRedeem instead
     */
    function redeem(
        uint256 /*shares*/,
        address /*receiver*/,
        address /*owner*/
    ) public virtual override returns (uint256) {
        revert("Redeem disabled: use requestRedeem");
    }

// ...existing code...
    /**
     * @notice Returns the maximum shares that can be minted
     * @dev Always returns 0 as mint is disabled
     * @param receiver Address that would receive the shares
     * @return Always 0 (mint disabled)
     */
    function maxMint(address /*receiver*/) public view virtual override returns (uint256) {
        return 0; // Mint is disabled
    }
}
```

Button: FIxed in commit 9d8ed75

Cyfrin: Verified.

7.4.2 Missing cancellation of withdrawal and redeem requests

Description: The withdrawal process in `BasisTradeVault` is a two-step mechanism. A user first calls `requestWithdraw` or `requestRedeem`, which places their request into a queue and escrows their vault shares within the contract. The second step, `processWithdrawal`, which actually sends the underlying assets to the user, can only be executed by a privileged address with the `AGENT_ROLE`.

This design introduces a significant centralization risk. If the agent(s) become malicious, are compromised, or simply stop performing their duties, they can refuse to call `processWithdrawal`. As a result, all pending withdrawal requests will be stuck in the queue indefinitely.

Users who have requested a withdrawal have their shares locked in the contract and have no way to unilaterally cancel their request to reclaim their shares. This means their funds are effectively frozen, entirely dependent on the liveness and cooperation of the agent.

Recommended Mitigation: To mitigate this, a function should be introduced that allows users to cancel their own pending withdrawal requests. This function would return the escrowed shares to the user, effectively un-staking them from the withdrawal queue.

A time-lock could be added to this cancellation function, such that a user can only cancel their request after a certain amount of time has passed since the request was made. This prevents users from spamming the queue while still providing an escape hatch if the agent is unresponsive.

Button: Acknowledged, will leave as is. Cancellation is tricky because the vault may have already incurred a cost in an attempt to wind down the basis trade and process the withdrawal. Did not add the ability to modify withdrawals to see real patterns in production for failed requests. In particular, do not think it makes sense to assist a user that is otherwise blacklisted from an ERC20 contract in recovering said assets

7.4.3 Consider using exponential notation in tests

Description: Tests frequently write decimals amounts as `100 * 10**6`. Consider using scientific notation (`100e6`) instead as it's more concise, improves readability (fewer visual tokens/zeros), and reduces exponent mistakes.

Button: Fixed in commit 9d8ed75

Cyfrin: Verified.

7.4.4 Unused `Pocket::approve` function

Description: The `Pocket` contract defines an `approve` function, but this function is never invoked within the current system architecture. Maintaining unused or unreferenced code increases the protocol's attack surface, as the function may be misused in future upgrades, or create assumptions that are no longer valid.

Recommended Mitigation: If the approve function is not required, remove it entirely.

Button: Acknowledged. Will be used in a new contract.

7.4.5 Consider implementing `ERC4626::mint`

Description: `BasisTradeVault` currently disables `mint()` and `previewMint()` (both revert), which reduces interoperability with ERC-4626 tooling, routers, vault aggregators, and simulators that rely on the "mint for exact shares" flow (e.g., slippage-aware deposits or migrators). Consider implementing:

- `previewMint(shares)` that **grosses up** the net asset requirement by adding the deposit fee on top (so the return is the *gross* assets a user must provide to mint `shares`), and

- enabling `mint(shares, receiver)` so it uses the standard ERC-4626 path (which will call the new `previewMint`), while enforcing the same controls as `deposit` (whitelist/TVL cap). This preserves your fee semantics (fee extracted from the provided gross amount) and materially improves compatibility with existing ERC-4626 integrations and tooling.

```
/**
 * @notice Preview the *gross* assets required to mint `shares`
 * @dev We gross-up the net assets (from base ERC4626 math) by adding the deposit fee on top.
 */
function previewMint(uint256 shares) public view virtual override returns (uint256) {
    require(shares > 0, "Cannot mint 0 shares");

    // Net assets required by base ERC4626 math (what must end up in the vault)
    uint256 netAssets = super.previewMint(shares);

    // Fee is charged on top of the net assets
    uint256 fee = _calculateFeeAmount(netAssets, depositFeeBps);
    return netAssets + fee; // gross = net + fee
}

/**
 * @notice Mint `shares` to `receiver`, charging the deposit fee on top
 * @dev Enforces the same whitelist/TVL checks as `deposit`.
 *      `super.mint` will use `previewMint` (gross) and enforce `maxMint(receiver)`.
 */
function mint(uint256 shares, address receiver) public virtual override checkDepositWhitelist(receiver)
↪  returns (uint256 assets) {
    return super.mint(shares, receiver);
}

/**
 * @notice Maximum shares that can be minted for `receiver`
 * @dev Derived from the gross-asset TVL cap (`maxDeposit`) by converting gross→net (remove fee),
 *      then net→shares using base ERC4626 math. Uses Floor rounding to stay within cap.
 */
function maxMint(address receiver) public view virtual override returns (uint256) {
    // Respect deposit whitelist if enabled
    if (depositWhitelistEnabled && !depositWhitelist[receiver]) {
        return 0;
    }

    // `maxDeposit(receiver)` returns the remaining *gross* capacity (fee included)
    uint256 grossCap = maxDeposit(receiver);
    if (grossCap == 0) return 0;

    // Convert gross -> net by extracting the fee portion from the gross amount
    uint256 feeFromGross = _extractFeeFromTotal(grossCap, depositFeeBps);
    uint256 netCap = grossCap - feeFromGross;

    // Convert the net-asset capacity to shares
    return _convertToShares(netCap, Math.Rounding.Floor);
}
```

And tests for the same in `BasisTradeVault.t.sol` (added test for `maxDeposit` as well):

```
function test_MintNoFees() public {
    // Setup the vault completely (already has 0% fees)
    _setupComplete();

    // Setup alice for deposits
    vm.prank(admin);
    vault.addToDepositWhitelist(alice);
    vm.prank(admin);
```

```
    vault.setTvlCap(100000 * 10**6);

    // Test 1: Alice deposits 10,000 USDC with 0% fee
    uint256 depositShares = 10000 * 10**6;

    vm.startPrank(alice);
    usdc.approve(address(vault), depositShares);
    uint256 actualAssets = vault.mint(depositShares, alice);
    vm.stopPrank();

    // With 0% fee, expect exactly 1:1 ratio
    assertEq(actualAssets, 10000 * 10**6); // Exactly 10,000 shares
    assertEq(actualAssets, depositShares);
    assertEq(vault.balanceOf(alice), depositShares);
    assertEq(vault.totalSupply(), 1000 * 10**6 + actualAssets); // Initial 1000 + alice's 10,000

    // Test 2: Bob deposits 5,000 USDC - still 1:1 with no fees
    vm.prank(admin);
    vault.addToDepositWhitelist(bob);

    uint256 bobShares = 5000 * 10**6;

    vm.startPrank(bob);
    usdc.approve(address(vault), bobShares);
    uint256 bobActualAssets = vault.mint(bobShares, bob);
    vm.stopPrank();

    // Still expect 1:1 with no fees
    assertEq(bobActualAssets, 5000 * 10**6);
    assertEq(bobActualAssets, bobShares);
    assertEq(vault.balanceOf(bob), bobShares);
}

function test_MintWithFees() public {
    // Setup the vault completely
    _setupComplete();

    // Set deposit fee to 1%
    vm.prank(admin);
    vault.setDepositFee(100);

    // Add alice to whitelist and increase TVL cap
    vm.prank(admin);
    vault.addToDepositWhitelist(alice);
    vm.prank(admin);
    vault.setTvlCap(100000 * 10**6);

    // Check initial state
    assertEq(vault.depositFeeBps(), 100); // 1% fee
    assertEq(vault.totalSupply(), 1000 * 10**6); // Initial deposit
    assertEq(vault.totalAssets(), 1000 * 10**6); // Initial assets

    // Alice mints 10,000 shares
    uint256 sharesAmount = 1000 * 10**6;

    // Preview how many shares alice should get
    uint256 expectedAssets = vault.previewMint(sharesAmount);

    // Execute the deposit
    vm.startPrank(alice);
    usdc.approve(address(vault), expectedAssets);
    uint256 actualAssets = vault.mint(sharesAmount, alice);
    vm.stopPrank();
```

```
    // Verify shares minted
    assertEq(actualAssets, expectedAssets);
    assertEq(actualAssets, 1010 * 10**6); // Exact amount, roughly 100bps less
    assertEq(vault.balanceOf(alice), sharesAmount);

    // Verify total supply increased by shares minted
    assertEq(vault.totalSupply(), 1000 * 10**6 + sharesAmount);

    // Verify funds are still in the vault (not moved to pocket yet)
    assertEq(usdc.balanceOf(address(vault)), 1000 * 10**6 + actualAssets);
}

function test_MintNoFees() public {
    // Setup the vault completely (already has 0% fees)
    _setupComplete();

    // Setup alice for deposits
    vm.prank(admin);
    vault.addToDepositWhitelist(alice);
    vm.prank(admin);
    vault.setTvlCap(100000 * 10**6);

    // Test 1: Alice deposits 10,000 USDC with 0% fee
    uint256 depositShares = 10000 * 10**6;

    vm.startPrank(alice);
    usdc.approve(address(vault), depositShares);
    uint256 actualAssets = vault.mint(depositShares, alice);
    vm.stopPrank();

    // With 0% fee, expect exactly 1:1 ratio
    assertEq(actualAssets, 10000 * 10**6); // Exactly 10,000 shares
    assertEq(actualAssets, depositShares);
    assertEq(vault.balanceOf(alice), depositShares);
    assertEq(vault.totalSupply(), 1000 * 10**6 + actualAssets); // Initial 1000 + alice's 10,000

    // Test 2: Bob deposits 5,000 USDC - still 1:1 with no fees
    vm.prank(admin);
    vault.addToDepositWhitelist(bob);

    uint256 bobShares = 5000 * 10**6;

    vm.startPrank(bob);
    usdc.approve(address(vault), bobShares);
    uint256 bobActualAssets = vault.mint(bobShares, bob);
    vm.stopPrank();

    // Still expect 1:1 with no fees
    assertEq(bobActualAssets, 5000 * 10**6);
    assertEq(bobActualAssets, bobShares);
    assertEq(vault.balanceOf(bob), bobShares);
}

function test_MintWithFees() public {
    // Setup the vault completely
    _setupComplete();

    // Set deposit fee to 1%
    vm.prank(admin);
    vault.setDepositFee(100);

    // Add alice to whitelist and increase TVL cap
```

```
    vm.prank(admin);
    vault.addToDepositWhitelist(alice);
    vm.prank(admin);
    vault.setTvlCap(100000 * 10**6);

    // Check initial state
    assertEq(vault.depositFeeBps(), 100); // 1% fee
    assertEq(vault.totalSupply(), 1000 * 10**6); // Initial deposit
    assertEq(vault.totalAssets(), 1000 * 10**6); // Initial assets

    // Alice mints 10,000 shares
    uint256 sharesAmount = 1000 * 10**6;

    // Preview how many shares alice should get
    uint256 expectedAssets = vault.previewMint(sharesAmount);

    // Execute the deposit
    vm.startPrank(alice);
    usdc.approve(address(vault), expectedAssets);
    uint256 actualAssets = vault.mint(sharesAmount, alice);
    vm.stopPrank();

    // Verify shares minted
    assertEq(actualAssets, expectedAssets);
    assertEq(actualAssets, 1010 * 10**6); // Exact amount, roughly 100bps less
    assertEq(vault.balanceOf(alice), sharesAmount);

    // Verify total supply increased by shares minted
    assertEq(vault.totalSupply(), 1000 * 10**6 + sharesAmount);

    // Verify funds are still in the vault (not moved to pocket yet)
    assertEq(usdc.balanceOf(address(vault)), 1000 * 10**6 + actualAssets);
}
```

Button: Fixed by the Cyfrin team in commit `d38f046`

Cyfrin: Verified. `mint` is implemented.

7.4.6 Consider enabling on-behalf-of withdrawals

Description: `BasisTradeVault`'s withdrawal queue only supports self-initiated requests that always pay the requester (`receiver = msg.sender`). For better ERC-4626 alignment and composability (routers, managers, third party integrations), extend the existing queue API to accept an explicit `receiver` and persist caller / owner / receiver in the queue item. Then use these values when processing to transfer assets to `receiver` and emit the canonical ERC-4626 `Withdraw(caller, receiver, owner, assets, shares)` event correctly.

Consider extending the queued item to persist new addresses:

```
struct WithdrawalRequest {
    address owner;     // share owner whose shares were escrowed
    address receiver;  // recipient of assets on payout
    address caller;    // who enqueued the request
    uint256 shares;
    uint256 assets;
    uint256 timestamp;
}
```

Update existing event:

```
struct WithdrawalRequested {
    address owner;
    address receiver;
```

```
    address caller;
    uint256 shares;
    uint256 assets;
    uint256 timestamp;
}
```

and change the request function signatures together with consuming the allowance:

```
function requestWithdraw(
    uint256 assets,
    address receiver,
    address owner
) external returns (uint256 queuePosition) {
    require(assets > 0, "Cannot withdraw 0 assets");
    uint256 grossAssets = assets + _calculateFeeAmount(assets, withdrawalFeeBps);
    uint256 shares = _convertToShares(grossAssets, Math.Rounding.Ceil);
    return requestRedeem(shares, receiver, owner);
}

function requestRedeem(
    uint256 shares,
    address receiver,
    address owner
) public requirePocket returns (uint256 queuePosition) {
    address caller = msg.sender;

    require(receiver != address(0), "Invalid receiver");
    require(owner != address(0), "Invalid owner");
    require(shares > 0, "Cannot redeem 0 shares");

    // spend allowance when acting on behalf of `owner`
    if (caller != owner) {
        _spendAllowance(owner, caller, shares);
    }

    // ...

    emit WithdrawalRequested(caller, owner, receiver, assetsAfterFee, shares, fee, queuePosition);
}
```

And lastly use stored identities in processing

```
function processWithdrawal() external onlyAgent {
    require(queueHead < queueTail, "No withdrawals to process");

    uint256 currentQueuePosition = queueHead;
    WithdrawalRequest memory request = withdrawalQueue[currentQueuePosition];
    require(request.owner != address(0), "Invalid withdrawal request");

    // Ensure sufficient vault liquidity
    uint256 vaultBalance = IERC20(asset()).balanceOf(address(this));
    require(vaultBalance >= request.assets, "Insufficient vault balance for withdrawal");

    // Burn escrowed shares and advance queue
    _burn(address(this), request.shares);
    totalEscrowedShares -= request.shares;
    totalPendingWithdrawals -= request.assets;
    delete withdrawalQueue[currentQueuePosition];
    queueHead++;

    // Pay the correct receiver
    IERC20(asset()).safeTransfer(request.receiver, request.assets);
```

```
    // Emit canonical ERC-4626 Withdraw with correct identities
    emit Withdraw(request.caller, request.receiver, request.owner, request.assets, request.shares);
    emit WithdrawalProcessed(request.owner, request.assets, currentQueuePosition);
}
```

This change maintains current safety (shares are escrowed immediately), improves interoperability, and ensures events and payouts reflect the actual caller/owner/receiver roles.

Button: Implemented in commit 9cde24c

Cyfrin: Verified. Caller/owner/receiver now stored and used in the event as well as approvals used in `requestRedeem`.

7.4.7 Missing SPDX License Identifiers

Description: All Solidity source files are missing an SPDX license identifier. Add a license header at the very top of each file to clarify licensing and silence tooling warnings (Solc/linters/CI):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;
```

Without this, some tools treat files as unlicensed or emit warnings, which can hinder downstream reuse and compliance.

Button: Acknowledged. Have not decided our licensing yet, but will add these headers before we make the repo public.

7.4.8 `BasisTradeVault::totalAssets` may undercount pocket-held funds

Description: `BasisTradeVault::totalAssets` currently depends on the (not-yet-finalized) oracle to report the Pocket's balance. The oracle may miss pocket-held funds by excluding assets sitting in the Pocket, which can skew previews and caps. While this can be considered the oracle's responsibility, consider ensuring `totalAssets()` includes Pocket-held assets, either by including them explicitly: `IERC20(asset()).balanceOf(pocket)` or by making sure the development of the oracle will include them.

Button: Fixed in 9cde24c. Moved to a PPS based oracle.

Cyfrin: Verified. The oracle now just returns a price-per-share, which is used to determine the total assets.

7.4.9 Use named mapping parameters to explicitly note the purpose of keys and values

Description: Use named mapping parameters to explicitly note the purpose of keys and values:

- BasisTradeTailor

```
    // Mappings
    /// @notice Maps pocket address to the user who controls it
    mapping(address => address) public pocketUser;
    /// @notice Tracks pending withdrawal amounts for each pocket
    mapping(address => uint256) public withdrawalRequests;
    /// @notice Whitelist of addresses allowed to create pockets
    mapping(address => bool) public creationWhitelist;
```

- BasisTradeVault:

```
    mapping(address => bool) public depositWhitelist;
```

- PocketFactory:

```
    mapping(address => bool) public approvedTailors;
```

Button: Fixed in commit a9ba276

Cyfrin: Verified.

7.4.10 Remove obsolete return statements when using named return variables

Description: Remove either the named return value or the `return` statement.

- BasisTradeVault::requestWithdraw

```
function requestWithdraw(uint256 assets) external returns (uint256 queuePosition) {
    // ...

    return requestRedeem(shares);
}
```

- BasisTradeVault::

```
function requestRedeem(uint256 shares) public requirePocket returns (uint256 queuePosition) {
    // ...

    return queuePosition;
}
```

- Pocket::exec

```
function exec(address target, bytes calldata data) external onlyOwner returns (bytes memory
↪  result) {
    // ...

    return result;
}
```

Button: Fixed in commit 9d8ed75

Cyfrin: Verified.

7.5 Gas Optimization

7.5.1 Redundant `approve(0)` in `BasisTradeVault::depositToTailor`

Description: `BasisTradeVault::depositToTailor` grants `tailor` an allowance, calls `tailor.deposit(pocket, amount)`, and then sets the allowance back to zero:

```
IERC20(asset()).forceApprove(address(tailor), amount);
tailor.deposit(pocket, amount);
IERC20(asset()).forceApprove(address(tailor), 0); // redundant
```

`BasisTradeTailor::deposit` pulls exactly `amount` via `safeTransferFrom(msg.sender, pocket, amount)`, which consumes the entire allowance. With a standard ERC20, the post-call allowance is already `0`, so the trailing `forceApprove(..., 0)` performs an unnecessary storage write and external call.

Consider removing the final zeroing call.

Button: Fixed in commit `9d8ed75`

Cyfrin: Verified.