Button Update Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

BengalCatBalu

February 13, 2026

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

Button/BasisTrade is an asset router designed to bridge between the EVM and HyperLiquid HyperCore as well as Morpho vaults. A whitelisted creator (typically the team/ops) creates a per-user `Pocket` wallet via `BasisTradeTailor` and `PocketFactory`. The "pocket user" then deposits the configured `baseAsset` (e.g., USDC) directly into their Pocket through `BasisTradeTailor.deposit`. Withdrawals are two-step: the pocket user submits a withdrawal request (`requestWithdrawal`), and an agent later fulfills it by transferring the base asset out of the Pocket to the pocket user (`processWithdrawal`).

For trading and account management, the Tailor controls each Pocket (as owner) and can execute HyperCore actions by calling the CoreWriter system contract (`sendRawAction`) from the Pocket (encoded via `CoreWriterEncoder`). This includes adding API wallets, moving spot assets, transferring USD-class balances between spot/perp, and managing native HYPE staking flows (deposit/withdraw/delegate/undelegate and withdrawing HYPE to approved destinations).

Separately, the Tailor supports a protocol adapter execution path: registered adapters return a list of `Call`s (either External calls executed via `pocket.execWithValue` or Approve calls executed via `pocket.approve`). This enables integrations such as Morpho Blue via the `MorphoBlueAdapter`. In the Morpho integration, the adapter builds calls to the Morpho core contract (e.g., borrow/repay, collateral operations) and includes prerequisite ERC20 approvals, allowing the Pocket to open and manage Morpho positions on behalf of the Pocket.

4.2 Actors and Roles

- **1. Actors:**
  - **Button/BasisTrade team (ops):** Deploys and upgrades the Tailor (UUPS), configures supported assets/adapters, and manages whitelists/roles.
  - **Pocket users:** Can deposit base asset into their Pocket and create withdrawal requests.

  - **Agents** / **Engine:** Perform operational actions like moving funds to/from HyperCore, processing withdrawals, and transferring balances between spot/perp.
- **2. Roles (contract-level):**
  - **DEFAULT_ADMIN_ROLE (Tailor):** UUPS upgrade authority; can approve withdrawal destinations (for HYPE withdrawals) and manage roles.
  - **OPERATOR_ROLE (Tailor):** Manages configuration like pocket creation whitelist, supported assets/adapters, and executes HYPE staking operations.
  - **AGENT_ROLE (Tailor):** Executes HyperCore operations (e.g., API wallet management, spot transfers, USDC to CoreDepositWallet) and processes withdrawal requests.
  - **ENGINE_ROLE (Tailor):** Performs margin-management style actions like USD-class transfers between spot/perp.
  - **Pocket owner (Tailor):** Each `Pocket` is owned by the Tailor and only the Tailor can call `transfer/approve/exec/execWithValue/transferNative`.

4.3 Key Components

- **BasisTradeTailor (UUPSUpgradeable):** Central coordinator that (i) creates Pockets via `PocketFactory`, (ii) maps each Pocket to a `pocketUser`, (iii) supports user deposit + withdrawal request/processing, (iv) executes HyperCore actions via CoreWriter calls from the Pocket, and (v) executes registered protocol adapters by translating adapter `Call[]` into Pocket `execWithValue` / `approve` operations.
- **Pocket:** A minimal ownable smart-wallet (clone) controlled by the Tailor. It can hold ERC20s and native tokens, transfer tokens, force-approve spenders, execute arbitrary calls (with or without native value), and transfer native tokens out. It disables `renounceOwnership` to avoid accidental loss of control.
- **PocketFactory:** Deploys Pocket clones and initializes them with the Tailor as owner; tracks pockets created by the factory (via `isPocket`). Tailor creation is gated operationally through a whitelist on the Tailor side.
- **Adapters (MorphoBlueAdapter):** A pluggable integration mechanism where adapters generate and validate call bundles. Tailor executes each call via the Pocket, separating arbitrary external calls from structured approvals. `MorphoBlueAdapter` specifically encodes the Morpho Blue interaction sequence (approval + Morpho call) so the Pocket supplies/withdraws collateral and borrows/repays against Morpho markets while maintaining `onBehalf` (and intended `receiver`) semantics tied to the Pocket.

4.4 Centralization risk

Control is concentrated in the Tailor's upgrade/admin keys and role holders. `DEFAULT_ADMIN_ROLE` can upgrade the Tailor (UUPS), and privileged roles (Operator/Agent/Engine) can execute CoreWriter actions and invoke adapter-driven Pocket calls, which can move assets and change Pocket state. Operational safety depends on strict key management, multisig/timelock hygiene, and monitoring of privileged actions and configuration (supported assets/adapters, creation whitelist, and approved withdrawal destinations).

5 Audit Scope

The audit scope was limited to:

```
script/AddApiWallet.s.sol
script/AddAssetToBasisTradeTailor.s.sol
script/AddToCreationWhitelist.s.sol
script/ApproveAgentInBasisTradeTailor.s.sol
script/ApproveEngineInBasisTradeTailor.s.sol
script/ApproveTailorInFactory.s.sol
script/ApproveWithdrawalDestination.s.sol
script/DeployMorphoBlueAdapter.s.sol
script/DeployPocketFactory.s.sol
```

```
script/GrantEngineRoleInBasisTradeTailor.s.sol
script/GrantOperatorRoleInBasisTradeTailor.s.sol
script/GrantOperatorRoleInPocketFactory.s.sol
script/ProcessWithdrawal.s.sol
script/RegisterAdapterInBasisTradeTailor.s.sol
script/RemoveAdapterFromBasisTradeTailor.s.sol
script/RevokeWithdrawalDestination.s.sol
script/TransferAssetFromCore.s.sol
script/TransferAssetToCore.s.sol
script/TransferPerp.s.sol
script/UpgradeBasisTradeTailor.s.sol
src/BasisTradeTailor.sol
src/Pocket.sol
src/PocketFactory.sol
src/adapters/BaseAdapter.sol
src/adapters/MorphoBlueAdapter.sol
src/libraries/CoreWriterEncoder.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Button Update smart contracts provided by Button. In this period, a total of 14 issues were found.

During the audit we identified 1 medium-risk issue, several low-risk items, a handful of informational/hardening observations, and one gas optimization.

The medium-risk finding is a withdrawal-request race: `BasisTradeTailor::requestWithdrawal` overwrites an existing request, so if a user updates a pending request while an agent processes in the same block, transaction ordering can lead to both the old and new amounts being withdrawn (contrary to user intent).

The low-risk items are mostly edge cases and integration limitations: `PocketFactory`'s ERC-165 reporting is non-compliant; the first USDC transfer to an unactivated HyperCore account can lose ~1 USDC to activation fees; `transferHypeToCore` can truncate precision due to 18→8 decimal conversion; Merkl/Morpho rewards are not claimable by Pocket owners as integrated; and `addAsset` ties adapter approval registration to Core transfer enablement.

Informational notes are primarily hardening and ops clarity (events, comment mismatches, stricter adapter validation such as `receiver == pocket`, interface checks for approved Tailors, and script ergonomics/Safe-mode consistency).

The gas optimization is to batch Pocket interactions: `_transferUsdcToCore` currently does approve → deposit → reset as three Pocket calls, which can be done via a single batched Pocket entrypoint.

The Button team also did a formatting commit during the audit, `c352db7`, which was reviewed and deemed to be safe.

After the audit some refactorings were done:

- `8d40ba7` to use custom errors,
- `89b6c6c` enforcing lint and
- `975b704` with some minor cleanup.

All of these were reviewed and deemed safe.

6.1 Operational concerns

- **Staking timing and sequencing hazards (delegate/undelegate/withdraw):** HyperCore staking has a 1-day delegation lockup and a 7-day unstaking queue from staking → spot. This creates operational pitfalls:

(i) delegation/undelegation flows must be carefully ordered and timed, because certain sequences can effectively "reset" the undelegation/availability timeline if new delegation actions are performed while an undelegation/withdrawal is in progress; and (ii) after calling `requestStakingWithdraw` via the Tailor, funds are not immediately available on the Core spot balance and are only credited after the 7-day queue elapses, so withdrawals and rebalancing must account for this latency. More info in this article

- **Gas griefing risk around `withdrawHype` execution:** `withdrawHype` ultimately perforrom the Pocket to the destination (`Pocket::transferNative`). If the operational flow ever exposes user influence over gas limits or encourages arbitrary "sponsored execution" patterns, an attacker can grief by crafting transactions with tight gas or by forcing expensive execution paths, causing repeated reverts/timeouts and wasting operator gas. Operationally, this means being precise about the gas policy (who submits, with what gas settings) and monitoring for repeated failures.

Summary

Project Name	Button Update
Repository	button-protocol
Commit	15bb95edd97a...
Fix Commit	975b70442cd1...
Audit Timeline	Feb 2nd - Feb 6th, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	5
Informational	7
Gas Optimizations	1
Total Issues	14

Summary of Findings

[M-1] `BasisTradeTailor` withdrawal request overwrite enables race conditions	Resolved
[L-1] `PoketFactory` is ERC-165 non compilant	Resolved
[L-2] First USDC transfer to unactivated HyperCore account loses 1 USDC to activation fee	Resolved
[L-3] Decimal mismatch in `BasisTradeTailor:transferHypeToCore` causes precision loss	Resolved
[L-4] Morpho Blue Market Rewards Cannot Be Claimed by Pocket Owners	Acknowledged

[L-5] Dual-Purpose mapping forces permissions intersections	Resolved
[I-1] `Pocket::execWithValue` does not emit native transfer event	Resolved
[I-2] `BasisTradeTailor::transferPerp` comment mismatch	Resolved
[I-3] `BasisTradeTailor::coreDepositWallet` is not blocked for adapters calls	Resolved
[I-4] `MorphoBlueAdapter::validateCalls` does not enforce `receiver == pocket`	Resolved
[I-5] `PocketFactory::approveTailor` doesn't verify `ITailor` interface implementation	Resolved
[I-6] Upgrade script deploys implementation but doesn't execute upgrade	Resolved
[I-7] Adapter removal script lacks Safe-mode calldata output	Resolved
[G-1] Batch pocket calls in `BasisTradeTailor::_transferUsdcToCore`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 `BasisTradeTailor` withdrawal request overwrite enables race conditions

Description: The `BasisTradeTailor::requestWithdrawal` function unconditionally overwrites any existing withdrawal request:

```
// BasisTradeTailor.sol:556-560
function requestWithdrawal(address pocket, uint256 amount) external onlyPocketUser(pocket) {
    withdrawalRequests[pocket] = amount;  // Overwrites existing request
    emit WithdrawalRequested(pocket, amount);
}
```

There is no validation to prevent overwrites or explicit cancellation mechanism. Users attempting to modify their withdrawal amount create race conditions with the agent's `processWithdrawal()` calls.

Impact: Transaction ordering between user's `requestWithdrawal()` and agent's `processWithdrawal()` determines whether the request is replaced or accumulated, leading to users withdrawing more than intended.

When users call `requestWithdrawal()` to modify an existing request, they expect the new amount to **replace** the old amount. However, if the agent processes the original request first, the user's second call creates a **new** request instead of replacing the original, resulting in both amounts being withdrawn.

Proof of Concept:

```
Block N:
  User: requestWithdrawal(pocket, 100 baseAsset)
  withdrawalRequests[pocket] = 100

User realizes they want only 50 total, submits modification:

Block N+1 (both transactions in same block):
  User: requestWithdrawal(pocket, 50)    // User wants to REPLACE 100 with 50
  Agent: processWithdrawal(pocket, 100)  // Agent processes original request

Outcome depends on transaction order within block:

Case 1 - User tx executes first:
  1. withdrawalRequests[pocket] = 50 (replaced)
  2. Agent processes 50 baseAsset
  3. withdrawalRequests[pocket] = 0
  4. Result: User withdraws 50 total

Case 2 - Agent tx executes first:
  1. Agent processes 100 baseAsset, sets withdrawalRequests[pocket] = 0
  2. User sets withdrawalRequests[pocket] = 50 (creates NEW request)
  3. Agent later processes this 50 baseAsset request
  4. Result: User withdraws 150 total (100 + 50)

In Case 2, the user wanted to reduce their total withdrawal to 50 but received 150 due to transaction
↪  ordering.
```

Similar issue occurs when users call `requestWithdrawal(0)` to cancel - if the agent processes first, the full amount is withdrawn before cancellation takes effect.

Recommended Mitigation: Prevent changing from non-zero to non-zero by requiring explicit cancellation first. Add a separate `cancelWithdrawal()` function to set the request to zero:

```
function requestWithdrawal(address pocket, uint256 amount) external onlyPocketUser(pocket) {
    require(amount > 0, "Amount must be positive");
    require(withdrawalRequests[pocket] == 0, "Cancel existing request first");
```

```
    withdrawalRequests[pocket] = amount;
    emit WithdrawalRequested(pocket, amount);
}

function cancelWithdrawal(address pocket) external onlyPocketUser(pocket) {
    uint256 currentRequest = withdrawalRequests[pocket];
    require(currentRequest > 0, "No pending request");

    withdrawalRequests[pocket] = 0;
    emit WithdrawalCancelled(pocket, currentRequest);
}
```

This prevents overwrites (cannot go from 100 → 50 directly) and makes cancellation explicit (must call `cancelWithdrawal()` to set to zero, cannot use `requestWithdrawal(0)`).

Button: FIxed in commit 2aa92eb

Cyfrin: Verified. Recommendation implemented.

7.2 Low Risk

7.2.1 `PoketFactory` is ERC-165 non compilant

Description: `PocketFactory` implements `IPocketFactory` but `supportsInterface()` doesn't check for it:

```
// PocketFactory.sol:93-100
function supportsInterface(bytes4 interfaceId)
    public view override(AccessControlEnumerable) returns (bool)
{
    return super.supportsInterface(interfaceId);  // doesn't check IPocketFactory
}
```

This violates ERC-165 standard. Calling `pocketFactory.supportsInterface(type(IPocketFactory).interfaceId)` returns `false` even though the contract implements the interface.

Recommended Mitigation: Check for `IPocketFactory` interface explicitly:

```
function supportsInterface(bytes4 interfaceId)
    public view override(AccessControlEnumerable) returns (bool)
{
-   return super.supportsInterface(interfaceId);
+   return
+       interfaceId == type(IPocketFactory).interfaceId ||
+       super.supportsInterface(interfaceId);
}
```

Button: Fixed in commit b74a07d

Cyfrin: Verified.

7.2.2 First USDC transfer to unactivated HyperCore account loses 1 USDC to activation fee

Description: HyperCore accounts must be activated before they can receive spot transfers without fees. According to HyperLiquid documentation, the first inbound transfer to an unactivated account triggers activation, which charges a ~1 USDC fee automatically deducted from the transferred amount.

The `BasisTradeTailor::transferAssetToCore` function transfers USDC from pocket to HyperCore via CCTP without verifying the pocket's Core account is activated:

```
// BasisTradeTailor.sol:295-312
function transferAssetToCore(address pocket, uint64 tokenIndex, uint256 amount) external onlyAgent {
    require(pocketUser[pocket] != address(0), "Pocket does not exist");
    require(amount > 0, "Amount must be positive");

    AssetConfig memory assetConfig = supportedAssets[tokenIndex];
    require(assetConfig.tokenAddress != address(0), "Asset not supported");

    if (tokenIndex == USDC_TOKEN_INDEX) {
        _transferUsdcToCore(pocket, amount);  // No activation check
    } else {
        address systemAddress = CoreWriterEncoder.getTokenSystemAddress(tokenIndex);
        IPocket(pocket).transfer(assetConfig.tokenAddress, systemAddress, amount);
    }
    // ...
}
```

Within `_transferUsdcToCore()`:

```
// BasisTradeTailor.sol:319-335
function _transferUsdcToCore(address pocket, uint256 amount) internal {
    // Approve CoreDepositWallet
    bytes memory approveData = abi.encodeWithSelector(
```

```
        IERC20.approve.selector,
        coreDepositWallet,
        amount
    );
    IPocket(pocket).exec(usdcAddress, approveData);

    // Deposit to HyperCore via CCTP - no activation check!
    bytes memory depositData = abi.encodeWithSelector(
        ICoreDepositWallet.depositFor.selector,
        pocket,
        amount,
        uint32(type(uint32).max)
    );
    IPocket(pocket).exec(coreDepositWallet, depositData);
}
```

The L1Read provides a `coreUserExists(address user)` method to check activation status, but this is not used. Additionally, deployment scripts (`script/TransferAssetToCore.s.sol`) also do not verify activation before initiating transfers.

Impact: If a newly created pocket's Core account is not activated before the first USDC transfer:

1. Agent calls `transferAssetToCore(pocket, USDC_TOKEN_INDEX, 100e6)` expecting 100 USDC to arrive on Core
2. CCTP transfer via `coreDepositWallet.depositFor()` is initiated
3. HyperLiquid detects unactivated account and deducts ~1 USDC activation fee
4. Only ~99 USDC arrives in the pocket's Core spot account
5. No event or error indicates this fee was charged

Which could result in an unexpected 1 USDC loss on first transfer to each new pocket, with no on-chain indication this occurred. Only trusted `AGENT_ROLE` addresses can trigger transfers, and the fee amount is small (1 USDC per pocket). Operators can activate accounts off-chain before first production deposit to avoid the fee. However, there is no on-chain enforcement or script-based check, relying entirely on operational discipline.

Recommended Mitigation: Add `coreUserExists` precompile check in `transferAssetToCore()` or `_transferUsdcToCore()` to verify account activation before the first transfer. Either revert with a clear error message directing operators to activate the account off-chain first, or create a script that handles activation separately (sending 1.1 USDC to cover the fee) before production deposits begin.

Additionally, update `script/TransferAssetToCore.s.sol` to check activation status and warn operators if transferring to an unactivated account.

Button: Fixed in 8ca2df0

Cyfrin: Verified. A call `activateCoreAccount` was added to `BasisTradeTailor` to send an amount `> 1 USDC`.

7.2.3 Decimal mismatch in `BasisTradeTailor:transferHypeToCore` causes precision loss

Description: The `BasisTradeTailor::transferHypeToCore` function accepts `uint256 amount` with 18 decimals (HyperEVM standard), but when bridging to HyperCore, the amount is truncated to 8 decimals. Any precision beyond 8 decimals is permanently lost.

```
// BasisTradeTailor.sol:400-412
function transferHypeToCore(address pocket, uint256 amount) external onlyOperator {
    require(pocketUser[pocket] != address(0), "Pocket does not exist");
    require(amount > 0, "Amount must be positive");

    IPocket(pocket).transferNative(
        CoreWriterEncoder.HYPE_SYSTEM_ADDRESS,  // Bridges to Core with 8 decimals
        amount  // uint256 with 18 decimals - precision beyond 8 decimals lost
    );
```

```
    lastCoreActionBlock[pocket] = block.number;
    emit LastCoreActionBlockUpdated(pocket, block.number);
    emit HypeTransferredToCore(pocket, amount);  // Emits full 18-decimal amount
}
```

Per HyperLiquid documentation, HYPE uses 18 decimals on HyperEVM but only 8 decimals on HyperCore. The bridge automatically truncates any precision beyond 8 decimals.

Impact: Each bridge transaction loses the fractional amount beyond 8 decimal precision. While individual losses are small (~9e-10 HYPE per transaction in worst case), they accumulate over time and represent permanent fund loss.

Recommended Mitigation: Add validation requiring amounts to be multiples of `1e10` (8 decimal precision):

```
function transferHypeToCore(address pocket, uint256 amount) external onlyOperator {
    require(pocketUser[pocket] != address(0), "Pocket does not exist");
    require(amount > 0, "Amount must be positive");
    require(amount % 1e10 == 0, "Amount must be multiple of 1e10 for 8-decimal precision");

    IPocket(pocket).transferNative(CoreWriterEncoder.HYPE_SYSTEM_ADDRESS, amount);

    lastCoreActionBlock[pocket] = block.number;
    emit LastCoreActionBlockUpdated(pocket, block.number);
    emit HypeTransferredToCore(pocket, amount);
}
```

Button: Fixed in commit b74a07d

Cyfrin: Verified. Amount verified to be a multiple of `1e10`.

7.2.4 Morpho Blue Market Rewards Cannot Be Claimed by Pocket Owners

Description: Morpho Blue implements external reward distribution through Merkl, a third-party reward distribution service. Per Merkl documentation, rewards are distributed manually - users must call claim functions to receive their rewards.

Pocket owners have no mechanism to claim Morpho rewards earned by their pockets:

```
// Pocket.sol:92-110
function exec(address target, bytes calldata data)
    external
    onlyOwner
    returns (bytes memory result)
{
// Restricted only to interaction with CoreWriter
}
```

The MorphoBlueAdapter only supports core Morpho operations (supply, withdraw, borrow, repay) and does not include reward claiming.

Impact: Rewards earned through Morpho Blue market participation accumulate in Merkl's distribution contract but remain inaccessible to pocket users. According to Merkl documentation, some reward campaigns have claiming deadlines - unclaimed rewards may disappear if not processed before the deadline.

While Merkl offers address remapping as a workaround (allowing smart contracts that cannot claim to forward rewards to an EOA), this requires manual intervention: pocket owners must contact the Merkl team, provide proof of ownership, and request remapping for each pocket individually.

Additionally, the protocol could upgrade `BasisTradeTailor` to add reward claiming functionality, but this leaves existing rewards unclaimed until the upgrade is deployed and executed.

Recommended Mitigation: There are two viable approaches to address this issue:

Approach 1: Add Operator-Controlled Reward Claiming (Requires Contract Changes)

Implement a restricted function that allows OPERATOR or AGENT roles to claim rewards on behalf of pockets and transfer them to the pocket owner.

Approach 2: Merkl Address Remapping Monitoring (No Contract Changes)

As a simpler alternative that requires no contract modifications, the protocol team can:

1. Monitor Merkl reward distributions for all Morpho campaigns
2. Proactively use Merkl's address remapping feature to redirect rewards from pocket addresses (smart contracts) to the corresponding pocket owner EOAs
3. Automate this monitoring and remapping process as part of protocol operations

Button: Acknowledged, will not be addressing this. our current plans for this codebase is to have an external vault contract own an underlying pocket via tailor, unlikely that we have an immediate future for many retail pockets. if there are rewards to coordinate for the vault, we can take it on with the rewarding teams directly

7.2.5 Dual-Purpose mapping forces permissions intersections

Description: The `BasisTradeTailor::addAsset` function sets both `registeredAssets` (used for adapter approvals in `executeAdapter()`) and `supportedAssets` (used for Core transfers in `transferAssetToCore/FromCore`) simultaneously:

```
// BasisTradeTailor.sol:652-672
function addAsset(address asset, uint64 tokenIndex) external onlyOperator {
    registeredAssets[asset] = true;  // Used for executeAdapter approvals (line 796)
    supportedAssets[tokenIndex] = AssetConfig({
        asset: asset,
        tokenIndex: tokenIndex
    });  // Used for Core transfers
}
```

When using MorphoBlueAdapter, both loanToken and collateralToken must be approved via `executeAdapter()`, which requires them to be in `registeredAssets`. Since `addAsset()` sets both mappings together, enabling MorphoBlueAdapter automatically enables `transferAssetToCore()` and `transferAssetFromCore()` for those tokens.

Impact: The protocol cannot use MorphoBlueAdapter without enabling Core transfer operations for loanToken and collateralToken. This forced functionality addition means the OPERATOR cannot selectively enable Morpho lending without also allowing direct bridge transfers for those assets.

For example, the protocol might want to support USDC lending via Morpho but disable direct USDC transfers to Core. This configuration is impossible because `addAsset()` couples both permissions. The protocol loses granular control over which operations are permitted for each asset.

Recommended Mitigation: Separate the mappings to allow independent control for bridge and adapter functionality.

Button: Fixed in `9a1ad0a`

Cyfrin: Verified. Mappings now separated.

7.3 Informational

7.3.1 `Pocket::execWithValue` does not emit native transfer event

Description: The `Pocket::execWithValue` function sends native tokens via the `value` parameter but only emits `Executed(target, selector)`, not `NativeTransferred(to, amount)`:

```
// Pocket.sol:107
result = target.functionCallWithValue(data, value);  // sends native tokens

emit Executed(target, selector);  // doesn't include value amount
```

This differs from `transferNative()` which properly emits `NativeTransferred(to, amount)` (line 130). Off-chain systems tracking native token movements through events will miss transfers made via `execWithValue()`, since the `Executed` event doesn't include the `value` parameter.

Recommended Mitigation: Emit `NativeTransferred` when value is sent to maintain consistency with `transferNative()`

```
function execWithValue(...) external onlyOwner returns (bytes memory result) {
    require(target != address(0), "Invalid target");
    result = target.functionCallWithValue(data, value);

    bytes4 selector;
    if (data.length >= 4) {
        selector = bytes4(data[:4]);
    }

+   if (value > 0) {
+       emit NativeTransferred(target, value);
+   }
    emit Executed(target, selector);
}
```

Button: Fixed in commit b74a07d

Cyfrin: Verified.

7.3.2 `BasisTradeTailor::transferPerp` comment mismatch

Description: `BasisTradeTailor::transferPerp` NatSpec comment states "(agent only)" but the function uses `onlyEngine` modifier:

```
// BasisTradeTailor.sol:373-378
/**
 * @notice Transfer funds between spot and perp accounts on HyperCore (agent only)
 */
function transferPerp(address pocket, uint64 amount, bool toPerp) external onlyEngine {
    // Comment says "agent only" but modifier is onlyEngine
```

The code implementation is likely correct since `ENGINE_ROLE` can call only this function, making the comment misleading.

Recommended Mitigation: Update the comment to match the implementation:

```
/**
- * @notice Transfer funds between spot and perp accounts on HyperCore (agent only)
+ * @notice Transfer funds between spot and perp accounts on HyperCore (engine only)
 * @param pocket Address of the pocket
 * @param amount Amount to transfer
 * @param toPerp True to transfer to perp, false to transfer to spot
 */
function transferPerp(address pocket, uint64 amount, bool toPerp) external onlyEngine {
```

Button: Fixed in commit b74a07d

Cyfrin: Verified.

7.3.3 `BasisTradeTailor::coreDepositWallet` is not blocked for adapters calls

Description: `BasisTradeTailor::executeAdapter` blocks `coreWriter` but not `coreDepositWallet`:

```
// BasisTradeTailor.sol:781
require(target != coreWriter, "Adapter cannot call coreWriter");
// No check for coreDepositWallet
```

`CoreDepositWallet.depositFor(address user, ...)` can send pocket USDC to arbitrary Core addresses. While adapters are trusted, this creates inconsistency in Core contract restrictions - if `coreWriter` is blocked for defense-in-depth, `coreDepositWallet` (which also interacts with Core) should be blocked too.

Recommended Mitigation:

```
-require(target != coreWriter, "Adapter cannot call coreWriter");
+require(target != coreWriter && target != coreDepositWallet, "Adapter cannot call Core contracts");
```

Button: Fixed in commit b74a07d

Cyfrin: Verified.

7.3.4 `MorphoBlueAdapter::validateCalls` does not enforce `receiver == pocket`

Description `MorphoBlueAdapter` constructs Morpho calls with `receiver = pocket` (e.g., `borrow(..., receiver)` and `withdrawCollateral(..., receiver)`), but `validateCalls` only validates `onBehalf == pocket` and does not check that the decoded `receiver` parameter also equals `pocket`. This weakens the adapter's intended "calls are safe and self-contained to the Pocket" guarantee.

Consider require `receiver == pocket` in addition to `onBehalf == pocket` for applicable calls.

Button: Fixed in 877074a.

Cyfrin: Verified. Also added some validation of additional market params.

7.3.5 `PocketFactory::approveTailor` doesn't verify `ITailor` interface implementation

Description: The `PocketFactory.approveTailor()` function only validates that the tailor address is non-zero and has code, but doesn't verify it implements the `ITailor` interface required for pocket management operations. There is also no such check in `ApproveTailorInFactory.s.sol`

```
// PocketFactory.sol:56-62
function approveTailor(address tailor) external onlyRole(OPERATOR_ROLE) {
    require(tailor != address(0), "Invalid tailor address");
    require(tailor.code.length > 0, "Tailor must be a contract");  // Only checks has code

    approvedTailors[tailor] = true;
    emit TailorApproved(tailor);
}
```

Recommended Mitigation: Add an ERC-165 interface check in `approveTailor()`. Also consider adding this check to the `ApproveTailorInFactory.s.sol` deployment script as an additional safety layer.

Button: Fixed in commit b74a07d

Cyfrin: Verified.

7.3.6 Upgrade script deploys implementation but doesn't execute upgrade

Description: The `UpgradeBasisTradeTailor.s.sol` script deploys a new implementation but doesn't execute the actual upgrade - line 44 is commented out.

```
// UpgradeBasisTradeTailor.s.sol:38-48
// Deploy new implementation
BasisTradeTailor newImpl = new BasisTradeTailor(hypeTokenIndex, usdcAddress, coreDepositWallet);
console.log("New implementation deployed at:", address(newImpl));
console.log("HYPE Token Index:", hypeTokenIndex);
console.log("USDC Address:", usdcAddress);
console.log("CoreDepositWallet:", coreDepositWallet);

//tailor.upgradeToAndCall(address(newImpl), "");  // Commented out

vm.stopBroadcast();

console.log("\n=== Tailor Upgraded ===");  // Misleading - upgrade didn't happen
```

The script logs "Tailor Upgraded" but the proxy still points to the old implementation. The `runSafe()` function (lines 60-83) suggests upgrades are meant for Gnosis Safe, but the `run()` function's behavior could confuse operators expecting a complete upgrade.

Recommended Mitigation: Either uncomment line 44 to execute the upgrade, or update documentation and console logs to clarify this is "deploy-only" mode and upgrade must be executed separately via Safe or manual `upgradeToAndCall()`.

Button: Fixed in commit `b74a07d`

Cyfrin: Verified.

7.3.7 Adapter removal script lacks Safe-mode calldata output

Description `RemoveAdapterFromBasisTradeTailor.s.sol` does not follow the `SAFE_MODE` pattern used in other operational scripts (i.e., printing `to/value/data` calldata for multisig execution) and instead relies on direct execution flow.

Consider add a `SAFE_MODE` path that prints the encoded calldata (`to`, `value`, `data`) for `removeAdapter(adapter)`.

Button: Fixed in commit `b74a07d`

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Batch pocket calls in `BasisTradeTailor::_transferUsdcToCore`

Description `BasisTradeTailor::_transferUsdcToCore` currently performs three separate `IPocket.exec(...)` calls: approve USDC, call `depositFor`, then reset approval back to `0`. Each `exec` incurs repeated external-call overhead.

Consider adding a batched call function to `Pocket` (e.g., `call(Call[] calldata calls)`), and update `_transferUsdcToCore` to perform the approve, depositFor, reset sequence via a single Pocket call:

- `IPocket.sol`:

```
struct Call {
    address target;
    bytes data;
    uint256 value;
}

function exec(Call[] calldata calls) external returns (bytes[] memory results);
```

- `Pocket.sol`:

```
function exec(Call[] calldata calls) external onlyOwner returns (bytes[] memory results) {
    uint256 len = calls.length;
    results = new bytes[](len);

    for (uint256 i = 0; i < len; ++i) {
        Call memory call = calls[i];
        if (call.value == 0) {
            results[i] = exec(call.target, call.data);
        } else {
            results[i] = execWithValue(call.target, call.data, call.value);
        }
    }
}
```

- `BasisTradeTailor._transferUsdcToCore`:

```
IPocket.Call calls = new IPocket.Call[3];
calls[0] = IPocket.Call({ target: usdcAddress, data: approveData, value: 0 });
calls[1] = IPocket.Call({ target: coreDepositWallet, data: depositData, value: 0 });
calls[2] = IPocket.Call({ target: usdcAddress, data: resetApproveData, value: 0 });

IPocket(pocket).call(calls);
```

Button: Fixed in `4122fe7`

Cyfrin: Verified.