

# Button

## Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:

January 16th to 20th, 2026

AUDITED BY:

Matte  
adriro

**Contents**

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	About Zenith	3
1.2	Disclaimer	3
1.3	Risk Classification	3
<hr/>		
<b>2</b>	<b>Executive Summary</b>	<b>3</b>
2.1	About Button	4
2.2	Scope	4
2.3	Audit Timeline	5
2.4	Issues Found	5
<hr/>		
<b>3</b>	<b>Findings Summary</b>	<b>5</b>
3.1	Medium Risk	7
3.2	Informational	9

## 1

## Introduction

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at <https://zenith.security>.

---

## 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

---

## 1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

# 2

## Executive Summary

### 2.1 About Button

Button makes it easy to do more with your Bitcoin. At the touch of a Button you can borrow, trade, or earn while still holding your Bitcoin.

---

### 2.2 Scope

The engagement involved a review of the following targets:

<b>Target</b>	button-protocol
<b>Repository</b>	<a href="https://github.com/buttonxyz/button-protocol">https://github.com/buttonxyz/button-protocol</a>
<b>Commit Hash</b>	f04d5b16c4fbffcbd27030ee00f3e5c882d5f616
<b>Files</b>	Diff from a349cf2c58fecaebe40c85f0cc9cb7819e6b8870 src/BasisTradeTailor.sol src/Pocket.sol src/PocketFactory.sol src/adapters/BaseAdapter.sol src/adapters/MorphoBlueAdapter.sol src/libraries/CoreWriterEncoder.sol script/DeployBasisTradeTailor.s.sol script/DeployPocketFactory.s.sol script/DeployMorphoBlueAdapter.s.sol script/DeployFraxlendAdapter.s.sol script/UpgradeBasisTradeTailor.s.sol

## 2.3 Audit Timeline

<b>January 16, 2026</b>	Audit start
<b>January 20, 2026</b>	Audit end
<b>February 2, 2026</b>	Report published

---

## 2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	0
Informational	9
<b>Total Issues</b>	<b>10</b>

# 3

## Findings Summary

ID	Description	Status
M-1	providePocketAllowance() allows approval of arbitrary spenders	Resolved
I-1	lastTransferBlock updated for non-EVM/Core transfers	Resolved
I-2	State variables usdcAddress and coreDepositWallet could be immutable	Resolved
I-3	Missing asset registration check in providePocketAllowance()	Resolved
I-4	transferHypeToCore() could use transferNative() for clarity	Resolved
I-5	executeAdapter() does not support native HYPE value in calls	Resolved
I-6	MorphoBlueAdapter withdrawCollateral() receiver parameter is redundant	Resolved
I-7	MorphoBlueAdapter uses raw approve() instead of SafeERC20	Resolved
I-8	Upgrade script should not be used with current implementation	Acknowledged
I-9	Redundant balance check in Pocket execWithValue()	Resolved

## 3.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] `providePocketAllowance()` allows approval of arbitrary spenders

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [BasisTradeTailer](#)

### Description:

The intended behavior is that sensitive outbound flows are constrained by explicit allowlists (e.g., adapter registration and withdrawal destination approval). In contrast, `providePocketAllowance` lets an AGENT\_ROLE address grant ERC20 approvals from any pocket to any spender without a comparable allowlist. That means a privileged agent can silently authorize arbitrary spenders, and the pocket's tokens become transferable via `transferFrom` without further on-chain checks.

A concrete scenario: assume a pocket holds 2,750 USDC. An agent calls `providePocketAllowance(pocket, usdc, attackerSpender, 2_750e6)`. From that point, the attacker's spender contract can pull the full balance via `transferFrom` even though the pocket user never approved that spender. This bypasses the system's usual flow controls (like `approvedWithdrawalDestinations` or adapter registration), because allowance is a separate ERC20 capability.

The vulnerable flow is short but powerful: `providePocketAllowance` validates only non-zero addresses and pocket existence, then delegates to `IPocket.approve`, which uses `forceApprove` on the token. There is no guard that the spender belongs to a vetted list, nor any token-specific restriction, so the effective policy is "any agent may grant any spender any allowance." The approvals persist until explicitly revoked, so a single call can create long-lived, silent access.

```
function providePocketAllowance( ... ) external onlyAgent {  
    require(pocketUser[pocket] != address(0), "Pocket does not exist");  
    require(spender != address(0), "Invalid spender"); // only a
```

```

    zero-address check
    IPocket(pocket).approve(token, spender, amount); // grants arbitrary
    spender authority
}

```

## Recommendations:

Enforce a spender allowlist (or reuse an existing authorization scheme) before granting approvals, mirroring the defensive pattern used for adapters and withdrawal destinations.

```

diff --git a/src/BasisTradeTailor.sol b/src/BasisTradeTailor.sol
index 000000..000000 100644
-- a/src/BasisTradeTailor.sol
++ b/src/BasisTradeTailor.sol
@@ -77,6 +77,8 @@ contract BasisTradeTailor is Initializable,
    AccessControlEnumerableUpgradeable,
    /// @notice Mapping to track which addresses are registered as assets
    mapping(address => bool) public registeredAssets;
    /// @notice Whitelist of addresses approved as withdrawal destinations
    mapping(address destination => bool approved)
    public approvedWithdrawalDestinations;
    /// @notice Whitelist of addresses approved as ERC20 allowance spenders
    mapping(address spender => bool approved) public approvedSpenders;

@@ -713,6 +715,7 @@ contract BasisTradeTailor is Initializable,
    AccessControlEnumerableUpgradeable,
    ) external onlyAgent {
        require(pocketUser[pocket] != address(0), "Pocket does not exist");
        require(token != address(0), "Invalid token");
        require(spender != address(0), "Invalid spender");
        require(approvedSpenders[spender], "Spender not approved");

        IPocket(pocket).approve(token, spender, amount);
    }

```

**Button:** Resolved by removing providePocketAllowance in [@6dcdccf037...](#)

**Zenith:**

## 3.2 Informational

A total of 9 informational findings were identified.

### [I-1] lastTransferBlock updated for non-EVM/Core transfers

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

#### Target:

- [BasisTradeTailor.sol](#)

#### Description:

The `lastTransferBlock` mapping is intended to track when assets are transferred between HyperEVM and HyperCore. However, this value is updated in several functions that do not perform actual cross-chain transfers:

- `transferPerp()` - transfers between spot and perp accounts within Core
- `depositToStaking()` - moves HYPE from spot to staking account within Core

```
// In transferPerp()
lastTransferBlock[pocket] = block.number;
emit LastTransferBlockUpdated(pocket, block.number);

// In depositToStaking()
lastTransferBlock[pocket] = block.number;
emit LastTransferBlockUpdated(pocket, block.number);
```

This creates semantic inconsistency where the variable name suggests EVM<->Core transfers, but the actual tracking includes Core-internal operations.

#### Recommendations:

Either rename the variable to reflect its broader purpose (e.g., `lastOperationBlock`), or remove these updates from functions that don't perform cross-chain transfers.

**Button:** Verified with [@23bea3cd8a...](#)

**Zenith:** Verified.

## [I-2] State variables `usdcAddress` and `coreDepositWallet` could be immutable

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [BasisTradeTailor.sol](#)

### Description:

The state variables `usdcAddress` and `coreDepositWallet` are set once during `initialize()` and never modified afterward. These could be declared as `immutable` or `constant`.

```
/// @notice USDC token address on HyperEVM
address public usdcAddress;

/// @notice CoreDepositWallet contract for USDC transfers to HyperCore
address public coreDepositWallet;
```

### Recommendations:

If these values are known at deployment time, consider moving them to constructor parameters and marking them as `immutable` or `constant`. Alternatively, document why they are storage variables if there's a design reason for keeping them mutable.

**Button:** Resolved with [@1346a4265d5...](#)

**Zenith:** Verified.

## [I-3] Missing asset registration check in `providePocketAllowance()`

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [BasisTradeTailor.sol](#)

### Description:

The `providePocketAllowance()` function allows approving any token address without verifying it's a registered asset. Other functions that operate on tokens through the pocket (such as `transferAssetToCore()`) validate the token against the `supportedAssets` mapping.

```
function providePocketAllowance(
    address pocket,
    address token,
    address spender,
    uint256 amount
) external onlyAgent {
    require(pocketUser[pocket] != address(0), "Pocket does not exist");
    require(token != address(0), "Invalid token");
    require(spender != address(0), "Invalid spender");
    // No check: require(registeredAssets[token], "Asset not registered");

    IPocket(pocket).approve(token, spender, amount);
}
```

### Recommendations:

Add a validation check to ensure the token is registered:  
`require(registeredAssets[token], "Asset not registered")`. This maintains consistency with other token operations in the contract.

**Button:** Resolved by removing `providePocketAllowance` in [@6dcdccf037...](#)

**Zenith:** Verified.

## [I-4] transferHypeToCore() could use transferNative() for clarity

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [BasisTradeTailor.sol](#)

### Description:

The `transferHypeToCore()` function transfers native HYPE to HyperCore by calling `execWithValue()` with empty calldata to `HYPE_SYSTEM_ADDRESS`. While this works with OpenZeppelin's `functionCallWithValue()`, as the system address has non-empty code, it would be better to issue a simple transfer.

```
IPocket(pocket).execWithValue(  
    CoreWriterEncoder.HYPE_SYSTEM_ADDRESS,  
    "",  
    amount  
)
```

The Pocket contract has a dedicated `transferNative()` function designed for simple native token transfers, which would be more semantically clear and explicit about the intent.

### Recommendations:

Consider using  
`IPocket(pocket).transferNative(CoreWriterEncoder.HYPE_SYSTEM_ADDRESS, amount)` for improved code clarity and explicit intent.

**Button:** Resolved with [@b1cccd2904a...](#)

**Zenith:** Verified.

## [I-5] executeAdapter() does not support native HYPE value in calls

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [BasisTradeTailor.sol](#)

### Description:

The `executeAdapter()` function iterates over adapter calls and executes them via `IPocket(pocket).exec()`, which does not forward any native value. This means adapters cannot build calls that require sending native HYPE.

```
for (uint256 i = 0; i < calls.length; i++) {
    lastResult = IPocket(pocket).exec(calls[i].target, calls[i].callData);
}
```

While the current adapters (MorphoBlueAdapter, FraxlendAdapter) don't require native value, this limitation would prevent future adapters from interacting with protocols that require HYPE transfers.

### Recommendations:

Consider extending the `IAdapter.Call` struct to include a `value` field and use `IPocket(pocket).execWithValue(calls[i].target, calls[i].callData, calls[i].value)` for execution. This would enable adapters to interact with protocols requiring native token transfers.

**Button:** Resolved with [@b1ccd2904a...](#)

**Zenith:** Verified.

## [I-6] MorphoBlueAdapter withdrawCollateral() receiver parameter is redundant

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [MorphoBlueAdapter.sol](#)

### Description:

The `withdrawCollateral()` function accepts a `receiver` parameter, but the `validateCalls()` function enforces that the receiver must be the pocket address itself. This makes the separate receiver parameter unnecessary.

```
function withdrawCollateral(address pocket, uint256 collateralAmount,
    address receiver)
    external
    view
    override
    returns (Call[] memory calls)
{
    // ...
    calls[0] = Call({
        target: morpho,
        callData: abi.encodeWithSelector(
            IMorpho.withdrawCollateral.selector,
            getMarketParams(),
            collateralAmount,
            pocket,
            receiver // Must equal pocket per validateCalls()
        )
    });
}
```

Since validation will reject any receiver that isn't the pocket, the function could simply use `pocket` directly for the receiver and remove the parameter.

**Recommendations:**

Remove the receiver parameter and use pocket directly in the call encoding, or document why the parameter exists if there's a future use case for allowing different receivers.

**Button:** Resolved with [@1346a4265d...](#) and made a similar change in [@c6e05c2bfc...](#)

**Zenith:** Verified.

## [I-7] MorphoBlueAdapter uses raw approve() instead of SafeERC20

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [MorphoBlueAdapter.sol](#)

### Description:

The MorphoBlueAdapter builds approval calls using the raw `IERC20.approve.selector` instead of SafeERC20's `forceApprove()`. Some non-standard ERC20 tokens (like USDT) may not return a boolean value from `approve()`, which could cause the call to revert when decoded.

```
calls[0] = Call({
    target: _collateralToken,
    callData: abi.encodeWithSelector(IERC20.approve.selector, morpho, 0)
});

calls[1] = Call({
    target: _collateralToken,
    callData: abi.encodeWithSelector(IERC20.approve.selector, morpho,
    collateralAmount)
});
```

The adapter does reset approvals to zero before setting a new value (handling tokens like USDT that require this pattern), but incompatible tokens with non-standard return values could still cause issues.

### Recommendations:

Since the adapter returns calls to be executed by the pocket rather than executing them directly, consider documenting which tokens are compatible or implementing a SafeERC20-aware approval pattern if non-standard tokens are expected to be supported.

**Button:** Resolved with [@ebe3cdb1b7...](#) by calling `Pocket.approve` from the adapter execution code

**Zenith:** Verified.

## [I-8] Upgrade script should not be used with current implementation

SEVERITY: Informational

IMPACT: Informational

STATUS: Acknowledged

LIKELIHOOD: Low

### Target:

- [UpgradeBasisTradeTailor.s.sol](#)

### Description:

The UpgradeBasisTradeTailor script exists in the repository but cannot be safely used with the current BasisTradeTailor implementation due to:

1. **Incompatible storage layouts:** Changes to state variables between versions could corrupt storage
2. **Missing re-initializer:** New state variables would not be properly initialized after an upgrade

Using this script without proper migration logic could brick the proxy or leave the contract in an inconsistent state.

### Recommendations:

The current in-scope BasisTradeTailor should be deployed from scratch, as it is incompatible with an upgrade scheme. Consider removing the script to prevent accidental use, or documenting the requirements to use it.

**Button:** Acknowledged.

## [I-9] Redundant balance check in Pocket execWithValue()

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

### Target:

- [Pocket.sol](#)

### Description:

The `execWithValue()` function includes an explicit balance check before calling OpenZeppelin's `functionCallWithValue()`, which performs the same check internally.

```
function execWithValue(
    address target,
    bytes calldata data,
    uint256 value
) external onlyOwner returns (bytes memory result) {
    require(target != address(0), "Invalid target");
    require(address(this).balance >= value, "Insufficient native
balance"); // Redundant

    result = target.functionCallWithValue(data, value);
    // ...
}
```

The OpenZeppelin `Address.functionCallWithValue()` already reverts with `AddressInsufficientBalance` if the contract doesn't have enough native balance.

### Recommendations:

Remove the redundant `require(address(this).balance >= value, ...)` check to save gas and reduce code duplication.

**Button:** Resolved with [@5f7ea0d1f...](#)

**Zenith:** Verified.