Brava Labs

# CCTP, ZeroEx, EIP712TypedDataSafeModule Integration

## Security Assessment Report

*Version: 2.0*

**November, 2025**

# Contents

# Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Brava Labs components in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

## Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

## Document Structure

The first section provides an overview of the functionality of the Brava Labs components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Brava Labs components in scope.

## Overview

The Brava protocol is a set of pools built on top of the Safe smart wallet platform that enable the use of self-custodied investment strategies, intended for consumption by institutional clients as well as less technical end-users.

Brava achieves this with a system of pools with delegated actions that act directly on each Safe smart wallet, avoiding the need to combine user funds or create undesirable trust assumptions.

This review covers Brava's integration with `CCTP`, `ZeroExSwap`, and a new `EIP712TypedDataSafeModule` to support offline transaction signing.

# Security Assessment Summary

## Scope

The review was conducted on the files hosted on the Brava Labs repository.

The scope of this time-boxed review was strictly limited to the following files, assessed at commit b3243a5. The fixes of the identified issues were assessed at commit 6c4ab1e.

1. *SequenceExecutor.sol*
2. *EIP712TypedDataSafeModule.sol*
3. *FeeTakeSafeModule.sol*
4. *SafeSetupRegistry.sol*
5. *ZeroExSwap.sol*
6. *CCTPBridgeSend.sol*
7. *CCTPBundleReceiver.sol*
8. *GasRefundAction.sol*
9. *UpgradeAction.sol*
10. *SafeDeployment.sol*

*Note: third party libraries and dependencies were excluded from the scope of this assessment.*

## Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

## Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

## Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.

- Medium: 2 issues.
- Low: 11 issues.
- Informational: 3 issues.

## Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Brava Labs components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

# Summary of Findings

| ID | Description | Severity | Status |
|---|---|---|---|
| BRV3-01 | Bundle Transactions Never Execute Due To Incorrect Implementation | Critical | Resolved |
| BRV3-02 | The Gas Price May Be Manipulated By The Bundle Executor To Steal Funds | High | Resolved |
| BRV3-03 | Users May Manipulate A Bundle To Avoid Gas Refund Payment | High | Closed |
| BRV3-04 | Under Compensation Of Executor's Resources Due To Incorrect Gas Accounting | Medium | Resolved |
| BRV3-05 | Malicious Users Can Drain Executor Funds Via Gas Griefing Attack | Medium | Resolved |
| BRV3-06 | Insufficient Token Transfer Validation In `GasRefundAction` | Low | Resolved |
| BRV3-07 | `GasRefundAction` Incompatibility With ERC20 Tokens Without Decimals Function | Low | Closed |
| BRV3-08 | `GasRefundAction` ETH/USD Oracle Incompatibility With Non-USD Pegged Tokens | Low | Closed |
| BRV3-09 | Missing Message Parsing And Validation In `CCTPBundleReceiver` | Low | Resolved |
| BRV3-10 | Missing USDC Token Validation In `CCTPBridgeSend` | Low | Closed |
| BRV3-11 | Persistent Token Allowance After Completed Swap | Low | Resolved |
| BRV3-12 | Missing Token Pair Validation | Low | Closed |
| BRV3-13 | Insufficient `swapCallData` Validation | Low | Closed |
| BRV3-14 | Inadequate Parameter Validation When Sending To CCTP | Low | Closed |
| BRV3-15 | `GasRefundAction` Multiple Silent Failure Points Reduce Transparency | Low | Resolved |
| BRV3-16 | Integer Underflow In Gas Calculation | Low | Resolved |
| BRV3-17 | Ignored Nonce Return Value From `depositForBurnWithHook()` | Informational | Resolved |
| BRV3-18 | Redundant Action Address Check In `EIP712TypedDataSafeModule` | Informational | Closed |
| BRV3-19 | Miscellaneous General Comments | Informational | Closed |

| **BRV3-01** | Bundle Transactions Never Execute Due To Incorrect Implementation | | |
|---|---|---|---|
| Asset | `CCTPBundleReceiver.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Critical | Impact: High | Likelihood: High |

## Description

The `CCTPBundleReceiver` contract contains an architectural issue that prevents bundle transactions from ever executing on the destination chain.

The current implementation incorrectly assumes that Circle's `MessageTransmitterV2` will automatically call back to the `handleReceiveFinalizedMessage()` function after processing a CCTP message. However, this callback mechanism does not exist in Circle's CCTP protocol, making the entire bundle execution flow impossible.

When users bridge USDC using `CCTPBridgeSend` with bundle data, the following occurs:

1. USDC is successfully burned on the source chain with hook data containing the bundle.
2. The hook data is embedded in the CCTP message sent to Circle.
3. On the destination chain, `relayReceive()` calls Circle's `MessageTransmitterV2.receiveMessage()` which mints USDC.
4. However, the `MessageTransmitterV2` does not call back to `handleReceiveFinalizedMessage()` as expected.
5. The bundle execution logic in `handleReceiveFinalizedMessage()` is never triggered.
6. Users receive their bridged USDC but their intended bundle transactions are never executed.

This can be verified by examining Circle's reference implementation at `CCTPHookWrapper.sol` which shows the correct pattern: after calling `receiveMessage()`, the hook data must be manually extracted from the message body using `BurnMessageV2._getHookData(_msgBody)` and then processed.

The vulnerability affects all CCTP bridge transactions that include bundle data, as the architectural assumption is fundamentally incorrect. The following check in `handleReceiveFinalizedMessage()` will never pass because Circle's `MessageTransmitterV2` never makes this call:

```
require(msg.sender == MESSAGE_TRANSMITTER, "CCTPBundleReceiver: Only MessageTransmitter can call");
```

## Recommendations

Restructure the `CCTPBundleReceiver` implementation to follow Circle's reference pattern:

1. Modify the `relayReceive()` function to:
   - Call `MessageTransmitter.receiveMessage()` to mint USDC.
   - Extract hook data from the message body manually using appropriate message parsing.
   - Process the extracted hook data by executing the bundle.

2. Implement message parsing logic to extract hook data from CCTP messages, similar to Circle's `BurnMessageV2._getHookData(_msgBody)` approach.
3. Move the bundle execution logic from `handleReceiveFinalizedMessage()` into the enhanced `relayReceive()` function.

The corrected flow should be:

```
function relayReceive(bytes calldata message, bytes calldata attestation) external returns (bool success) {
    // 1. Process CCTP message and mint USDC
    bool ok = IMessageTransmitterV2(MESSAGE_TRANSMITTER).receiveMessage(message, attestation);
    require(ok, "Message relay failed");

    // 2. Extract hook data from message
    bytes memory hookData = extractHookDataFromMessage(message);

    // 3. Execute bundle if hook data present
    if (hookData.length > 0) {
        _executeBundleFromHookData(hookData);
    }

    return ok;
}
```

It is worth noting that executing hook data along with USDC minting in one function call may pose some risks. For example, if the commands in the hook data always revert then the USDC minting also fails.

## Resolution

The issue has been addressed in commits c45d0e90, aad7f604, eea906d1, 9ed90581.

| BRV3-02 | The Gas Price May Be Manipulated By The Bundle Executor To Steal Funds | | |
|---|---|---|---|
| Asset | `GasRefundAction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: High | Impact: High | Likelihood: Medium |

## Description

Using `tx.gasprice` to compute executor refunds enables an attacker to artificially increase the refund. Under EIP-1559 and proposer–builder separation (PBS), a block producer (or any party that can control the fee recipient) can set an abnormally high priority fee and then recoup that tip themselves, while the contract refunds the inflated `tx.gasprice` in stablecoins. This drains funds from safes without corresponding economic cost to the attacker beyond the base fee.

Refunds are paid by `GasRefundAction.executeAction()` via `delegatecall` from Safe wallets to a designated executor (either `tx.origin` or `FEE_RECIPIENT`). The refund is computed in Stablecoins using the observed gas usage and `tx.gasprice`:

```
refundAmount = ((gasUsed * tx.gasprice) * uint256(answer)) / (10 ** denomExp);
```

Under EIP-1559:

```
tx.gasprice = min(tx.maxFeePerGas, block.basefee + tx.maxPriorityFeePerGas)
```

The priority fee (tip) is paid to the block's fee recipient. With PBS and common MEV flows, the effective fee recipient can be controlled by the builder/validator. This creates an incentive and an avenue to inflate the priority fee.

The contract assumes `tx.gasprice` is an honest proxy for the executor's economic cost. That assumption is unsafe because:

- A block producer (or colluding executor with influence over the fee recipient) can set an elevated `maxPriorityFeePerGas` so that `tx.gasprice = basefee + highTip`.
- The contract refunds `gasUsed * tx.gasprice` (converted via oracle), including the inflated tip.
- Economically, the attacker recovers their own tip as the fee recipient, so the true net cost is roughly `basefee * gasUsed`, while the refund pays out `(basefee + highTip) * gasUsed`.
- The difference (`tip * gasUsed`) is extracted from the safes in stablecoins, resulting in loss of funds.

This is not merely theoretical. In modern Ethereum with PBS, builders and validators regularly control the fee recipient. Similar dynamics also exist on many L2s, where the sequencer is the fee recipient and can influence `tx.gasprice` semantics.

Consider the following step-by-step exploit scenario:

1. Pre-conditions:
   - Gas refunds are enabled.
   - `GasRefundAction.executeAction()` computes refunds using `tx.gasprice`.
   - The attacker can ensure that the bundle executes in a block where the attacker (or a collaborator) is the fee recipient (builder/validator) or an L2 sequencer.
2. Action:

- Submit the bundle with very high `maxPriorityFeePerGas` so that `tx.gasprice` is large.
- The onchain refund calculation uses the inflated `tx.gasprice`, leading to a large stablecoin refund.

3. Economic Netting:
   - The attacker receives back the high tip as the fee recipient.
   - Only the base fee portion is a real external cost (burn).
   - `Profit = (tip * gasUsed * priceFactor)` paid out from safes in stablecoins.
4. Result:
   - Stablecoin drain proportional to the chosen tip and gas used, bounded only by any implicit gas usage limits and available safe balances.

## Recommendations

Implement a secure gas pricing mechanism that cannot be manipulated by transaction originators or fee recipients. For example, use a trusted external gas price oracle (e.g., Chainlink Fast Gas Price Oracle) instead of `tx.gasprice` for refund calculations

## Resolution

The issue has been addressed in commit 00195b48. The gas price oracle was implemented.

| **BRV3-03** | Users May Manipulate A Bundle To Avoid Gas Refund Payment | | |
|---|---|---|---|
| Asset | `GasRefundAction.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: High | Impact: Medium | Likelihood: High |

## Description

The system allows users to completely bypass gas refund payments by manipulating transaction bundles after signing.

The workflow involves Brava providing a bundle of transactions to users for signing, then users return the signed bundle back to Brava for execution. However, there is a critical flaw in this design where users can remove the command that executes `GasRefundAction.executeAction()` entirely from the bundle after signing it.

While `EIP712TypedDataSafeModule` validates that when `enableGasRefund=true`, at least one `FEE_ACTION` must be present in the signed data on lines [**228-233**], this validation only applies to the signed typed data structure. Once the user has signed the bundle with the required gas refund action included, they can modify the actual execution sequence before submitting it back to Brava, completely removing the gas refund execution.

The validation in `_validateSequenceActionsAndDetectRefund()` checks for the presence of refund actions, but this occurs on the data that was already signed, not on the actual execution sequence that will be processed by `SequenceExecutor`.

This creates a scenario where users can do the following set of actions:

1. Receive a bundle that includes the required gas refund action to pass validation.
2. Remove the gas refund action from the execution sequence.
3. Sign and submit the modified bundle for execution.
4. Receive execution services without paying the agreed gas refund.

## Recommendations

Implement a mechanism to cryptographically bind the raw bundle sent to the users to the signed bundle to prevent post-signature manipulation.

## Resolution

The issue was not applicable to Brava because Brava only accepts signatures from users.

| **BRV3-04** | Under Compensation Of Executor's Resources Due To Incorrect Gas Accounting | | |
|---|---|---|---|
| Asset | `EIP712TypedDataSafeModule.sol, GasRefundAction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Medium | Impact: Medium | Likelihood: Medium |

## Description

Executors receive insufficient gas refunds due to incorrect gas accounting in the gas refund system.

The current implementation fails to account for the complete gas consumption from transaction initiation to completion, since the executors use their funds to pay for this gas, they will frequently lose out on complete compensation leading to loss for the protocol or elimination of any incentives to execute bundles for executors.

The `consumeGasContext()` function returns `startGas` captured at the beginning of bundle execution on line [**171**] of `EIP712TypedDataSafeModule`, but this excludes gas consumed before the bundle starts executing:

```
// Record gas at the beginning and the executor for the refund action to consume later
gasStartBySafe[_safeAddr] = gasleft(); //@audit Gas recorded only at bundle start
executorBySafe[_safeAddr] = tx.origin;
```

In CCTP receive flows, significant gas consumption occurs in the `CCTPBundleReceiver` contract before bundle execution begins. The receiver processes the CCTP message, decodes hook data, and forwards the call to the EIP712 module all these consumes gas that remains unaccounted for in refund calculations.

Even with the `GAS_OVERHEAD` it is currently set to the minimum gas value not accounting for the gas used before the bundle's execution begins.

The gas refund action is not constrained to execute at the end of sequences. Documentation explicitly states: "*Position is not enforced; sequences should generally place fee actions at the end for clarity*" (`TYPED_DATA_MODULE.md:18`).

When actions execute after the gas refund action, their gas consumption cannot be included in the refund calculation since `consumeGasContext()` captures gas at the time of the refund action execution (`GasRefundAction.sol` lines [**94-95**])

```
uint256 gasUsed = startGas > gasleft() ? (startGas - gasleft() + GAS_OVERHEAD) : 0;
//@audit gasUsed only includes gas up to this point, not subsequent actions
```

This could lead to the following negative outcomes, potentially occurring on every transaction with gas refunds enabled:

- **Economic Impact**: executors systematically lose funds on each transaction.
- **Business Viability**: under compensation makes the executor business model unsustainable.
- **User Experience**: may lead to executor unavailability or increased service fees.

## Recommendations

Implement Full Transaction Gas Tracking - ensure The `GAS_OVERHEAD`accounts for all the gas used before gas tracking.

Enforce Gas Refund Action Positioning - ensure `_validateSequenceActionsAndDetectRefund()` of the `EIP712TypedDataSafeModule` contract checks that the gas refund actions is the last of the sequences whenever refunds are enabled.

### Resolution

The issue has been addressed in commit 00195b48. The gas measurement was moved to module entry and exit for a more accurate reading.

| BRV3-05 | Malicious Users Can Drain Executor Funds Via Gas Griefing Attack | | |
|---|---|---|---|
| Asset | `ZeroExSwap.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Medium | Impact: Medium | Likelihood: Medium |

## Description

Malicious users can exploit the `ZeroExSwap` contract to perform gas griefing attacks against executors, potentially draining their funds completely. The vulnerability stems from insufficient validation of the `swapTarget` parameter and the lack of gas consumption controls during swap execution.

The `ZeroExSwap` contract allows users to specify an arbitrary `swapTarget` address that receives the swap call with user-controlled `swapCallData`. While the contract validates that `swapTarget` is not the zero address on line [85], it does not implement an allowlist or validation mechanism to ensure the target is a legitimate 0x exchange contract.

The following snippet shows an example from `ZeroExSwap.sol` lines [84-85]:

```
// Basic validation - swap target cannot be zero address
require(_params.swapTarget != address(0), Errors.ZeroEx__InvalidSwapTarget(_params.swapTarget, address(0))); //@audit Insufficient
    ↪ validation
```

This insufficient validation creates a critical attack vector where malicious users can deploy contracts designed to consume maximum gas before reverting, causing executors to pay exorbitant gas fees while receiving no compensation.

Consider the following attack vectors:

1. **Malicious Contract Deployment**: Attacker deploys a contract with a function that intentionally consumes gas through expensive operations (loops, storage writes, external calls) before reverting.
2. **Bundle Creation**: Attacker creates a signed bundle containing:
   - A small legitimate USDC amount for bridging (to appear genuine)
   - A `ZeroExSwap` action with `swapTarget` pointing to the malicious contract
   - Gas-consuming `swapCallData` that triggers the expensive operations
3. **Executor Exploitation**: When the executor processes the bundle:
   - The malicious swap consumes significant gas before reverting
   - The executor pays full gas costs but cannot recover fees due to failed execution

Note the vulnerable call in `ZeroExSwap.sol` line [106]:

```
// Execute swap through 0x exchange contract
// Ensure forwarded ETH matches parameters for consistency
require(msg.value == _params.callValue, Errors.InvalidInput("ZeroExSwap", "callValueMismatch"));
require(_params.swapCallData.length > 0, Errors.InvalidInput("ZeroExSwap", "swapCallData"));
// solhint-disable-next-line avoid-low-level-calls
(bool success, ) = _params.swapTarget.call{value: msg.value}(_params.swapCallData); //@audit Arbitrary external call
if (!success) {
    revert Errors.ZeroEx__SwapFailed(); //@audit Revert after gas consumption
}
```

## Recommendations

Establish a curated `swapTarget` allowlist of legitimate 0x exchange contracts and proxy addresses.

This allowlist should be maintained by protocol administrators and updated as new 0x contract versions are deployed. The validation mechanism should verify that swap targets are known, audited 0x exchange contracts rather than arbitrary addresses controlled by users.

## Resolution

The issue has been addressed in commit 00195b48. A refund cap was implemented through `maxRefundAmount` to mitigate the issue.

| **BRV3-06** | Insufficient Token Transfer Validation In `GasRefundAction` | | |
|---|---|---|---|
| Asset | `GasRefundAction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Medium | Likelihood: Low |

## Description

The contract performs token transfers using a low-level call with insufficient validation of the transfer result.

The current implementation at `GasRefundAction.sol` on lines [**126-134**] uses a low-level call to transfer tokens:

```
(bool success, ) = p.refundToken.call(
    abi.encodeWithSelector(IERC20.transfer.selector, recipient, refundAmount)
);
if (success) {
    LOGGER.logActionEvent(
        LogType.GAS_REFUND,
        abi.encode(address(this), p.refundToken, refundAmount, recipient)
    );
}
```

This approach has several issues. Firstly, it only checks if the call succeeded, not whether the actual transfer was successful. Some ERC20 tokens return `false` instead of reverting on failure, while others may return `true` even if the transfer failed due to insufficient balance or other conditions. Additionally, the contract logs the refund as successful based solely on the call success, not the actual transfer success.

One potential exploitation scenario occurs when the user's Safe has insufficient token balance to pay the calculated refund amount. In standard ERC20 implementations, this would cause the transfer to revert, but the contract deliberately uses a low-level call to suppress these reverts. When the transfer fails due to insufficient funds, the function silently continues without reverting the transaction, emitting failure events, or providing any indication that payment failed. This creates an asymmetric risk where service providers (such as Brava) execute transactions expecting gas refund compensation, but users can consume services without payment when they lack sufficient token balances.

The vulnerability can be exploited when using non-standard ERC20 tokens that do not revert on failure but return `false`. In such cases, the contract would log a successful gas refund event even though no tokens were actually transferred to the recipient. This leads to incorrect accounting and creates confusion about whether refunds were actually processed.

Furthermore, some tokens implement transfer fees or deflationary mechanisms where the actual transferred amount differs from the requested amount. The current implementation cannot detect these scenarios, potentially leading to incorrect refund amount logging.

## Recommendations

Replace the low-level call with OpenZeppelin's SafeERC20 library to ensure proper token transfer validation and error handling.

SafeERC20 provides a `safeTransfer` function that automatically handles various ERC20 token implementations, including tokens that return `false` on failure instead of reverting, tokens with no return value, and tokens with non-standard behaviour. This library function will revert the transaction if the transfer fails for any reason, ensuring that gas refund events are only logged when tokens are actually transferred successfully.

## Resolution

The issue has been addressed in commit 00195b48. Token transfer now uses OpenZeppelin’s SafeERC20 library.

| **BRV3-07** | `GasRefundAction` Incompatibility With ERC20 Tokens Without Decimals Function | | |
|---|---|---|---|
| Asset | `GasRefundAction.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Medium | Likelihood: Low |

## Description

The gas refund system can be completely broken by using ERC20 tokens that do not implement the optional `decimals()` function.

The `GasRefundAction.executeAction()` function assumes all ERC20 tokens implement the `decimals()` function on `GasRefundActio`
line [**103**]:

```
uint8 td = IERC20Metadata(p.refundToken).decimals();
```

According to the EIP-20 specification, the `decimals()` function is marked as OPTIONAL in the ERC20 standard. Many legitimate ERC20 tokens, particularly older ones, do not implement this function.

When a refund token without a `decimals()` function is used, the contract call will revert, causing the entire transaction to fail. This creates several problematic scenarios:

1. **Complete system failure**: If the selected refund token lacks `decimals()`, all transactions requiring gas refunds will fail
2. **Unpredictable behaviour**: The system may work with some tokens but fail with others, creating inconsistent user experiences
3. **Lock-in risk**: Users may unknowingly select incompatible tokens, causing their transactions to become unusable

The issue occurs during refund calculation where token decimals are needed to properly convert between ETH gas costs and token amounts. Without proper handling, the contract cannot determine the appropriate decimal precision for the refund token.

## Recommendations

Implement safe handling for tokens that may not have a `decimals()` function:

1. **Use try-catch pattern**: Wrap the `decimals()` call in a try-catch block to handle tokens without this function.
2. **Static call validation**: Use a low-level static call to check if the function exists before calling it.
3. **Token registry enhancement**: Enhance the token registry to store decimal information for approved tokens, eliminating the need to call the function during execution.
4. **Fallback mechanism**: Implement a fallback to a reasonable default (such as 18 decimals) when the function is not available.

### Resolution

The issue is no longer relevant due to code refactoring.

| | |
|---|---|
| **BRV3-08** | `GasRefundAction` ETH/USD Oracle Incompatibility With Non-USD Pegged Tokens |
| Asset | `GasRefundAction.sol` |
| Status | **Closed:** See Resolution |
| Rating | Severity: Low / Impact: Medium / Likelihood: Low |

## Description

The contract relies solely on an ETH/USD oracle for gas refund calculations, which creates severe pricing inaccuracies when using non-USD-pegged tokens as refund tokens.

The `GasRefundAction` contract calculates gas refund amounts using only an ETH/USD price oracle on `GasRefundAction.sol` lines [**79-86**]. The refund calculation on `GasRefundAction.sol` lines [**100-109**] assumes that all refund tokens are effectively worth $1 USD per token unit:

```
uint8 od = ETH_USD_ORACLE.decimals();
uint8 td = IERC20Metadata(p.refundToken).decimals();
if (uint256(td) > 18 + uint256(od)) {
    return; // avoid underflow in exponent calculation
}
uint256 denomExp = 18 + uint256(od) - uint256(td);
refundAmount = ((gasUsed * tx.gasprice) * uint256(answer)) / (10 ** denomExp);
```

This calculation converts ETH gas costs to USD value using the ETH/USD oracle, then directly uses that USD amount as the number of refund tokens to transfer. This approach only works correctly for USD-pegged stablecoins such as USDC or USDT.

When non-USD-pegged tokens are used as refund tokens, the system produces drastically incorrect refund amounts:

1. **High-value tokens**: Using ETH ($4,500) or WBTC ($115,000) as refund tokens would result in massive overpayments. A $10 gas cost would incorrectly calculate as 10 ETH or 10 WBTC refunds worth $45,000 or $1,150,000 respectively.
2. **Low-value tokens**: Using tokens worth fractions of a dollar would result in severe underpayments, potentially making gas refunds economically meaningless.
3. **Volatile tokens**: Price fluctuations in non-USD-pegged tokens create unpredictable refund economics, potentially allowing exploitation during price movements.

The documentation on line [**48**] of `GAS_REFUND_SYSTEM.md` even acknowledges this limitation by recommending "*Prefer stablecoins as `refundToken`*", indicating awareness of the issue but without technical enforcement.

The `TokenRegistry` system provides no validation mechanism to ensure only USD-pegged tokens are approved for gas refunds, allowing administrators to inadvertently approve incompatible tokens.

Note, the `refundToken` must be explicitly added to the registry by an address holding the `TRANSACTION_EXECUTOR_ROLE`, meaning end users cannot freely choose arbitrary tokens. Since refunds are ultimately funded by user assets, administrators have a strong incentive to only approve stablecoins (e.g., USDC, USDT) that align with the expected $1 peg, hence the lower severity rating of this finding.

## Recommendations

Implement proper multi-token oracle support to handle non-USD-pegged refund tokens:

1. **Token-specific price oracles**: Extend the contract to support individual price feeds for each approved refund token. Modify the calculation to: `refundAmount = (gasUsed * tx.gasprice * ETH_USD_price) / TOKEN_USD_price`
2. **Enhanced TokenRegistry integration**: Extend the `TokenRegistry` to store oracle addresses alongside token approvals, ensuring each token has an associated price feed before approval.
3. **Calculation restructure**: Replace the current single-oracle calculation with:

```
// Get token-specific USD price
IAggregatorV3 tokenOracle = getTokenOracle(p.refundToken);
(, int256 tokenPrice, , uint256 tokenUpdatedAt, ) = tokenOracle.latestRoundData();

// Calculate refund: (ETH cost in USD) / (token price in USD)
refundAmount = ((gasUsed * tx.gasprice) * uint256(ethUsdPrice)) / uint256(tokenPrice);
```

4. **Fallback mechanism**: Implement validation to restrict refunds to USD-pegged tokens if proper oracle infrastructure cannot be established for all supported tokens.
5. **Administrative controls**: Add token type classification in the approval process to explicitly identify and handle USD-pegged versus non-USD-pegged tokens differently.

## Resolution

The issue is no longer relevant due to code refactoring.

| **BRV3-09** | Missing Message Parsing And Validation In `CCTPBundleReceiver` | | |
|---|---|---|---|
| Asset | `CCTPBundleReceiver.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `CCTPBundleReceiver` contract lacks message parsing and validation logic required by CCTP V2 protocol, as demonstrated in Circle's official `CCTPHookWrapper` reference implementation.

The contract makes several incorrect assumptions about CCTP message structure and hook data format. Even if the fundamental architectural flaw were fixed, the contract would still fail to correctly process CCTP messages due to missing message parsing capabilities:

1. **No Message Structure Parsing**: The contract does not parse the CCTP message structure to extract hook data. Circle's implementation uses `TypedMemView` library and message parsing functions:

   ```
   bytes29 _msg = message.ref(0);
   MessageV2._validateMessageFormat(_msg);
   bytes29 _msgBody = MessageV2._getMessageBody(_msg);
   bytes29 _hookData = BurnMessageV2._getHookData(_msgBody);
   ```

2. **No Version Validation**: The contract lacks message version and message body version validation that Circle requires:

   ```
   require(MessageV2._getVersion(_msg) == supportedMessageVersion, "Invalid message version");
   require(BurnMessageV2._getVersion(_msgBody) == supportedMessageBodyVersion, "Invalid message body version");
   ```

3. **Incorrect Hook Data Format**: The contract assumes hook data is ABI-encoded `executeBundle` call at `CCTPBundleReceiver.sol` lines [**87-88**], but Circle's specification requires:

   - First 20 bytes: target address
   - Remaining bytes: hook call data

   This fundamental format mismatch means the contract cannot correctly process CCTP hook data even if it received it.

4. **Insufficient Length Validation**: line [**81**] only validates `hookData.length > 0`, whilst Circle's reference performs proper length checking before processing hook data.

5. **Incorrect Hash Calculation**: line [**77**] calculates hash of `messageBody` instead of the full message, causing incorrect event tracking.

## Recommendations

1. Implement proper CCTP message parsing following Circle's specification:

   - Add message structure parsing libraries or equivalent functionality
   - Extract hook data using proper CCTP message format parsing
   - Validate message and body versions against supported versions

2. Align hook data format with CCTP specification:
   - Parse first 20 bytes as target address
   - Use remaining bytes as call data
   - Adapt bundle execution logic to work with correct format
3. Add comprehensive validation:

```
function parseMessage(bytes calldata message) internal pure returns (bytes memory hookData) {
    // Parse and validate message structure
    // Extract and validate message version
    // Extract message body and validate version
    // Extract hook data from message body
    // Validate hook data length
    return hookData;
}
```

4. Calculate message hash using the full message, not just `messageBody`.

## Resolution

The issue has been addressed in commits eea906d1, 9ed90581. Message parsing and validation were implemented.

| **BRV3-10** | Missing USDC Token Validation In `CCTPBridgeSend` | | |
|---|---|---|---|
| Asset | `CCTPBridgeSend.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `CCTPBridgeSend` contract accepts any token address as `usdcToken` parameter without validating that it is actually a legitimate USDC token address.

The issue manifests in both execution functions at `CCTPBridgeSend.sol` lines [**91-96**] and `CCTPBridgeSend.sol` lines [**120-124**] where the contract only performs basic zero address checks:

```
require(cctpParams.usdcToken != address(0), "Invalid USDC token address");
```

However, Circle's CCTP protocol only supports bridging USDC tokens. If users provide non-USDC token addresses, the transaction will fail when the `TOKEN_MESSENGER` contract attempts to process the bridge operation, but only after the Safe has already approved the token spend.

This creates a poor user experience where users can construct seemingly valid transactions that will always fail, potentially wasting gas and creating confusion. The contract name and documentation suggest it is specifically for USDC bridging, but the implementation does not enforce this constraint.

## Recommendations

Add explicit validation to ensure the provided token address is a recognised USDC token for the current chain.

## Resolution

The issue was acknowledged by the development team with the following comment:

> *"USDC address is passed as parameter to support multiple chains. Circle's TokenMessenger validates token compatibility; invalid tokens will revert at Circle's layer. Additional validation would duplicate Circle's checks. Furthermore, we intentionally maintain flexibility for future CCTP protocol expansion—Circle may support additional bridgeable tokens beyond USDC. Hardcoding USDC-specific validation would require contract upgrades to support such additions."*

| **BRV3-11** | Persistent Token Allowance After Completed Swap | | |
|---|---|---|---|
| Asset | `ZeroExSwap.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The contract uses `safeIncreaseAllowance()` to grant approval to the 0x Allowance Target but never revokes unused allowance after swap completion.

The vulnerability occurs at `ZeroExSwap.sol` line [96] where the contract executes the following command:

```
tokenIn.safeIncreaseAllowance(ALLOWANCE_TARGET, _params.fromAmount)
```

Since actions execute via `delegatecall` from the Safe wallet context, allowances from the Safe wallet to the `ALLOWANCE_TARGET` accumulate permanently over multiple transactions.

If the 0x swap consumes less than the approved amount, the remaining allowance persists indefinitely. Over time, this creates an expanding attack surface where accumulated allowances from multiple swaps could be exploited if the `ALLOWANCE_TARGET` were to be compromised.

Examination of the codebase shows that none of the other contracts using `safeIncreaseAllowance()` implement allowance cleanup, indicating this is a systemic pattern rather than an isolated issue.

## Recommendations

Implement allowance cleanup after swap execution to prevent persistent allowances:

```
// After swap completion at line 119
uint256 remainingAllowance = tokenIn.allowance(address(this), ALLOWANCE_TARGET);
if (remainingAllowance > 0) {
    tokenIn.safeDecreaseAllowance(ALLOWANCE_TARGET, remainingAllowance);
}
```

## Resolution

The issue has been addressed in commit 2370015. The allowance was reset to zero through `forceApprove()`.

| **BRV3-12** | Missing Token Pair Validation | | |
|---|---|---|---|
| Asset | `ZeroExSwap.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The contract does not validate that the input and output tokens are different, allowing meaningless same-token swaps.

The contract `ZeroExSwap` accepts `tokenIn` and `tokenOut` parameters in the `Params` struct but performs no validation to ensure these addresses are different. This allows users to execute swaps where `tokenIn` equals `tokenOut`, which serves no functional purpose.

Such transactions would:

- Consume gas for no beneficial outcome
- Generate misleading swap events in logs
- Potentially be used to game gas refund mechanisms
- Create confusion in transaction analysis

While the impact is low as it primarily affects user experience and gas efficiency rather than security, it represents a gap in input validation that could lead to user errors or be exploited for griefing purposes.

## Recommendations

Add validation to ensure input and output tokens are different, e.g.:

```
function _zeroExSwap(Params memory _params) internal {
    require(
        _params.fromAmount != 0 && _params.minToAmount != 0,
        Errors.InvalidInput(protocolName(), "executeAction")
    );

    // Add token pair validation
    require(
        _params.tokenIn != _params.tokenOut,
        Errors.InvalidInput(protocolName(), "sameTokenSwap")
    );

    // ... rest of implementation
}
```

## Resolution

The issue has been acknowledged by the development team with the following comment:

*"The 0x API does not provide same-token swap quotes under normal operation, making this an extremely unlikely scenario. Adding on-chain validation for every swap would impose unnecessary gas overhead for a minimal edge case that has no material security impact. Frontend validation prevents any accidental same-token swaps in the legitimate flow."*

| **BRV3-13** | Insufficient `swapCallData` Validation | | |
|---|---|---|---|
| Asset | `ZeroExSwap.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The contract performs minimal validation on user-provided swap calldata, only checking that the length is greater than zero.

The vulnerability occurs at `ZeroExSwap.sol` line [**104**] where the contract validates `swapCallData` with only the following code:

```
require(_params.swapCallData.length > 0, Errors.InvalidInput("ZeroExSwap", "swapCallData"))
```

This validation is insufficient as it does not verify:

1. Function selectors to ensure only legitimate swap functions are called.
2. Parameter structure to prevent malformed calls.
3. Target contract compatibility with the provided calldata.

The calldata is then used directly in an external call at `ZeroExSwap.sol` line [**106**]:

```
\code{_params.swapTarget.call{value: msg.value}(_params.swapCallData)
```

This combination of minimal validation with direct external calls could potentially allow:

1. Calls to unintended functions on the swap target.
2. Malformed function calls that might have unexpected behaviour.
3. Exploitation of edge cases in target contract implementations.

While the likelihood is low due to the requirement for sophisticated attack construction, the medium impact stems from the potential for unintended contract interactions that could affect swap outcomes or contract state.

## Recommendations

Implement function selector validation to restrict calls to known 0x swap functions.

## Resolution

The issue has been acknowledged by the development team with the following comment:

*"Extensive calldata validation would require maintaining a whitelist of function selectors across all 0x contract versions and chains. This would create maintenance burden as 0x regularly deploys new contracts and updates functions. Our security model relies on:*

- *Users sign transactions explicitly via EIP-712, providing informed consent*
- *Actions execute in Safe's delegatecall context*
- *The swapTarget validation (non-zero address) prevents null calls*
- *Off-chain validation ensures calldata matches signed intent"*

| **BRV3-14** | Inadequate Parameter Validation When Sending To CCTP | | |
|---|---|---|---|
| Asset | `CCTPBridgeSend.sol` | | |
| Status | **Closed:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The contract performs insufficient validation of critical CCTP parameters, specifically the USDC token address and destination caller address.

The contract validates that both parameters are non-zero but does not verify them against known valid values for the respective chains and domains. This creates two distinct but related validation gaps:

- **USDC Token Address Validation** (`CCTPBridgeSend.sol` line [92] and `CCTPBridgeSend.sol` line [120]) - the contract only checks that the USDC address is not zero:

  ```
  require(cctpParams.usdcToken != address(0), "Invalid USDC token address");
  ```

  However, CCTP exclusively works with official Circle USDC tokens. Each supported blockchain has a specific, fixed USDC address that is compatible with CCTP operations. Using an incorrect token address could lead to transaction failures or, in worst-case scenarios, burning the wrong token and causing loss of funds.

- **Destination Caller Validation** (`CCTPBridgeSend.sol` line [178] - similarly, the contract only validates that the destination caller is not zero:

  ```
  require(cctpParams.destinationCaller != bytes32(0), "Invalid destination caller");
  ```

  The CCTP protocol requires that this parameter matches the exact address authorised to call `receiveMessage()` on the destination chain. If set incorrectly, the cross-chain minting process will fail permanently, effectively locking the bridged funds. In the Brava protocol context, this should typically be the address of the `CCTPBundleReceiver` contract deployed on the destination chain.

While the likelihood is low because experienced teams typically configure these correctly, and the `TokenMessenger` provides some protection against invalid tokens, the lack of explicit validation creates unnecessary risk.

## Recommendations

Implement sufficient validation of critical CCTP parameters to ensure they are correct for the intended operation.

## Resolution

The issue has been acknowledged by the development team with the following comment:

> *"Similar to BRV3-10, Circle's protocol provides primary validation. Frontend validation prevents most user errors. Adding extensive on-chain validation for every chain/domain combination would significantly increase gas costs*

*for marginal security benefit. Additionally, maintaining flexibility for CCTP protocol expansion is important—as Circle adds support for new chains and potentially new tokens, hardcoded validation would require contract upgrades to accommodate these additions."*

| **BRV3-15** | `GasRefundAction` Multiple Silent Failure Points Reduce Transparency | | |
|---|---|---|---|
| Asset | `GasRefundAction.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `GasRefundAction` contract has numerous silent failure points that provide no visibility into why gas refunds may not be processed, creating operational opacity.

The contract contains multiple conditions that cause silent returns without any logging, events, or error indication. These silent failure points occur at the following locations, skipping refunds:

- line [**76**]: Token not approved by `TOKEN_REGISTRY`
- line [**88**]: Oracle price is non-positive
- line [**91**]: Oracle data is stale (over 1 hour old)
- line [**97**]: No gas was consumed (`gasUsed == 0`)
- line [**106**]: Token decimals would cause underflow
- line [**115**]: Calculated refund amount is zero
- line [**122**]: Recipient address is zero

Additionally, failed token transfers are not logged (`GasRefundAction.sol` lines [**126-134**]). Only successful transfers generate log events, while failures are completely invisible.

This design creates several operational problems:

1. **Debugging difficulty**: When refunds fail, there is no indication of the root cause
2. **Monitoring challenges**: Service providers cannot detect when they are not receiving expected compensation
3. **User confusion**: Users may expect refunds that never materialise with no explanation
4. **Audit trail gaps**: Failed refund attempts leave no records for compliance or analysis

The lack of transparency makes it difficult to identify systemic issues, troubleshoot problems, or provide users with explanations for missing refunds.

## Recommendations

Implement comprehensive event logging for all failure conditions:

1. **Add failure events**: Create specific events for each failure condition with descriptive reasons:

```
event GasRefundSkipped(address indexed safe, string reason, bytes additionalData);
event GasRefundFailed(address indexed safe, address token, uint256 amount, string reason);
```

2. **Log silent returns**: Replace silent returns with event emissions that include the reason for skipping the refund.
3. **Enhanced transfer logging**: Log both successful and failed transfer attempts with specific error details.
4. **Monitoring dashboard**: Consider implementing off-chain monitoring to track refund success rates and common failure reasons.
5. **User notifications**: Consider mechanisms to inform users when their refunds fail and why.

## Resolution

The issue has been addressed in commit 00195b48. The module now emits `GasRefundProcessed` event with status. While silent failures in adapter oracle checks are by design.

| **BRV3-16** | Integer Underflow In Gas Calculation | | |
|---|---|---|---|
| Asset | `SequenceExecutor.sol` | | |
| Status | **Resolved:** See Resolution | | |
| Rating | Severity: Low | Impact: Low | Likelihood: Low |

## Description

The `_executeActionStandard()` function performs arithmetic subtraction without checking for underflow when calculating gas to forward in `delegatecall`.

In `SequenceExecutor.sol` line [**102**], the code executes `sub(gas(), 5000)` in assembly context. When `gas()` returns less than 5000, this operation underflows and wraps around to produce a very large number (approximately 2ˆ256 - (5000 - `gas()`)). While the EVM's gas forwarding mechanism prevents this from causing immediate transaction failure, it disrupts the intended gas reservation pattern.

The likelihood is low because reaching this code path with less than 5000 gas remaining requires specific timing and gas manipulation. The impact is low because EIP-150 automatically limits gas forwarding to available gas minus 1/64th, preventing the underflow from causing catastrophic failure. The main consequence is that less gas than intended (only ~1/64th instead of 5000) remains for post-call operations.

## Recommendations

Add an underflow check before the subtraction.

## Resolution

The issue has been addressed in commit 00195b48. The gas calculation was updated.

| BRV3-17 | Ignored Nonce Return Value From `depositForBurnWithHook()` |
|---|---|
| Asset | `CCTPBridgeSend.sol` |
| Status | **Resolved:** See Resolution |
| Rating | Informational |

## Description

The contract fails to capture and utilise the nonce return value from the `TokenMessengerV2.depositForBurnWithHook()` function, which impacts transaction traceability and operational monitoring capabilities.

The `depositForBurnWithHook()` function in the `ITokenMessengerV2` interface returns a `uint64 nonce` that uniquely identifies each cross-chain message within Circle's CCTP infrastructure. This nonce serves as a critical identifier that links burn events on the source chain with corresponding mint events on the destination chain.

In the current implementation at `CCTPBridgeSend.sol` lines [**176-178**], the contract uses a low-level call to invoke the CCTP function but discards the returned nonce value:

```
(bool success, bytes memory returnData) = address(TOKEN_MESSENGER).call(cctpCallData);
if (success) {
    // CCTP call succeeded - but nonce is ignored
}
```

According to Circle's CCTP V2 documentation, whilst the nonce is not required for the operational success of cross-chain transfers (as Circle's infrastructure handles message delivery independently), capturing this value significantly enhances transaction monitoring and debugging capabilities. The nonce enables developers and operators to correlate specific burn events with their corresponding mint events across chains, facilitating comprehensive transaction tracking and troubleshooting of cross-chain operations.

The absence of nonce capture reduces operational visibility and makes it more difficult to trace specific cross-chain transfers through Circle's infrastructure, particularly when investigating failed or delayed transfers.

## Recommendations

Decode the nonce from the return data and emit it in the `CCTPBridgeExecuted` event to enhance transaction traceability. Modify the event definition to include the nonce parameter and update the emission logic accordingly.

## Resolution

The issue has been addressed in commit aad7f604 by improving event emission. The nonce tracking is handled offchain via Iris API.

| **BRV3-18** | Redundant Action Address Check In `EIP712TypedDataSafeModule` |
|---|---|
| Asset | `EIP712TypedDataSafeModule.sol` |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

The function `_validateSequenceActionsAndDetectRefund()` contains a redundant check for action address validity in `EIP712TypedDataSafeModule.sol` lines [**354-355**].

The code checks if `actionAddr == address(0)` and reverts with `EIP712TypedDataSafeModule_ActionNotFound` error. However, this check is unreachable because the previous line calls `ADMIN_VAULT.getActionAddress(actionId)`, which already performs the same validation and reverts with `AdminVault_NotFound` if the action address is zero.

The AdminVault's `getActionAddress()` function includes:

```
require(actionAddress != address(0), Errors.AdminVault_NotFound("action", _actionId));
```

This means the transaction would revert before reaching the redundant check, making `EIP712TypedDataSafeModule.sol` lines [**354-355**] an unreachable code.

## Recommendations

Remove the redundant check on `EIP712TypedDataSafeModule.sol` lines [**354-355**] to improve code clarity and reduce gas costs

## Resolution

The issue has been acknowledged by the development team.

| **BRV3-19** | Miscellaneous General Comments |
|---|---|
| Asset | All contracts |
| Status | **Closed:** See Resolution |
| Rating | Informational |

## Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unused Helper Functions Increase Contract Complexity**

   ***Related Asset(s): CCTPBridgeSend.sol***

   The contract contains three external pure helper functions that are either unused or have limited usage, resulting in unnecessary code complexity and increased deployment costs.

   The contract implements three helper functions at `CCTPBridgeSend.sol` lines [**237-300**]:

   - `createFastTransferParams()`,
   - `createStandardTransferParams()`, and
   - `createHookParams()`.

   These functions are designed to create `CCTPParamsV2` structs with predefined parameter configurations for different transfer types.

   Analysis of the codebase usage reveals:

   - `createFastTransferParams()` is used only in test files (`test/actions/cctp/CCTP-Real.test.ts`)
   - `createStandardTransferParams()` is used only in test files (`test/actions/cctp/CCTP-Real.test.ts`)
   - `createHookParams()` is not used anywhere in the codebase

   Each helper function adds gas to the contract deployment cost and increases the contract's bytecode size. Whilst these functions are marked as `external pure` and do not pose direct security risks, they represent unnecessary complexity in the contract's public interface.

   The presence of unused or minimally-used utility functions can create confusion about the intended usage patterns and may lead developers to believe these functions are required for proper contract interaction. Additionally, maintaining unused code increases the overall maintenance burden and code review complexity.

   Consider removing or relocating the helper functions based on their usage patterns:

   - Remove `createHookParams()` entirely as it is not used anywhere in the codebase
   - For `createFastTransferParams()` and `createStandardTransferParams()`, consider either:
     - Moving them to a separate utility library contract that can be imported where needed
     - Keeping them if they provide significant value for frontend integration or developer experience

   If the functions are retained, ensure they are documented as utility functions for external integrators and consider adding comprehensive tests to verify their parameter configurations remain correct.

2. **Code Duplication In `CCTPBridgeSend` Validation Logic**

   ***Related Asset(s): CCTPBridgeSend.sol***

   Identical validation logic is duplicated between two execution functions, creating maintenance risks and code quality issues.

   The contract contains two entry points for executing CCTP bridge operations:

- `executeAction()` and
- `executeActionWithBundleContext()`.

Both functions implement nearly identical validation logic that checks the same parameters with the same requirements.

In `executeAction()` (`CCTPBridgeSend.sol` lines [**93-98**]):

```
require(cctpParams.usdcToken != address(0), "Invalid USDC token address");
require(cctpParams.amount > 0, "Bridge amount must be greater than 0");
require(cctpParams.destinationDomain != 0, "Invalid destination domain");
require(cctpParams.destinationCaller != bytes32(0), "Invalid destination caller");
require(signature.length > 0, "Bundle signature required for CCTP");
```

In `executeActionWithBundleContext()` (`CCTPBridgeSend.sol` lines [**119-127**]):

```
require(cctpParams.usdcToken != address(0), "Invalid USDC token address");
require(cctpParams.amount > 0, "Bridge amount must be greater than 0");
require(cctpParams.destinationDomain != 0, "Invalid destination domain");
require(cctpParams.destinationCaller != bytes32(0), "Invalid destination caller");
require(_signature.length > 0, "Bundle signature required for CCTP");
```

This duplication creates a maintenance risk where changes to validation logic in one function might not be applied to the other, potentially leading to inconsistent behaviour or security vulnerabilities if validation requirements change in the future.

Consider extracting the validation logic to a single private function that can be shared by both entry points:

```
function _validateCCTPParams(
    CCTPParamsV2 memory cctpParams,
    bytes memory signature
) private pure {
    require(cctpParams.usdcToken != address(0), "Invalid USDC token address");
    require(cctpParams.amount > 0, "Bridge amount must be greater than 0");
    require(cctpParams.destinationDomain != 0, "Invalid destination domain");
    require(cctpParams.destinationCaller != bytes32(0), "Invalid destination caller");
    require(signature.length > 0, "Bundle signature required for CCTP");
}
```

Then call this function from both entry points to ensure consistent validation across all execution paths.

## Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

## Resolution

The development team's responses to the raised issues above are as follows.

1. Acknowledged.
2. Acknowledged.

# Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

**Impact**

| | Low | Medium | High |
|---|---|---|---|
| High | Medium | High | Critical |
| Medium | Low | Medium | High |
| Low | Low | Low | Medium |

**Likelihood**

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

## References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].