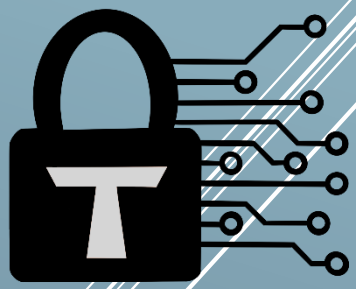
Trust Security

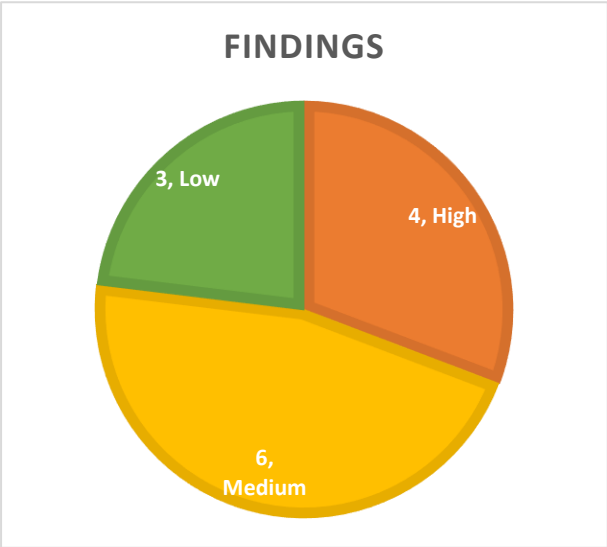


Smart Contract Audit

Brahma Console

15/05/2023

Executive summary

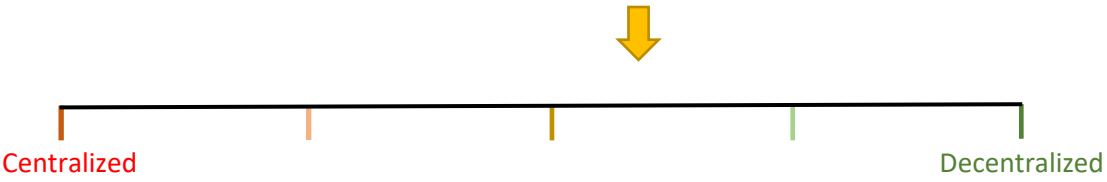


Category	Automation
Audited file count	36
Lines of Code	2194
Auditor	Trust
Time period	01/05/23 – 15/05/23

Findings

Severity	Total	Fixed	Acknowledged
High	4	3	1
Medium	6	4	2
Low	3	3	-

Centralization score



Signature

Digitally signed by Trust
Date: 2023-05-21 18:27:01

Document properties

Versioning

Version	Date	Description
0.1	14/05/23	Client report
0.2	21/05/23	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

Console-Core repository:

- src/*

Console-Integrations repository:

- src/*

Repository details

- **Repository URL:** https://github.com/brahma-fi/console-core/tree/ft-trusted-core
- **Commit hash:** 58bf05320bc5405f36549ca786a317724241e2ee
- **Mitigation review URL:** https://github.com/Brahma-fi/console-core-audit
- **Mitigation review hash:** d20dcacb6d2f4d2e64380717615cf9c7439e5063
- **Repository URL:** https://github.com/Brahma-fi/console-integrations/tree/ft-trusted-exec
- **Commit hash:** 630fdc4d3a21344c90c3fcda94ad045c6dbbe6dc
- **Mitigation review URL:** https://github.com/Brahma-fi/console-integrations-audit
- **Mitigation review hash**: c44caafacb4863f32f7457df49fe78bf8b6ced66

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project has maintained manageable complexity, reducing attack risks
Documentation	**Good**	Project is mostly well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Project has several concerning centralization risks.

Findings

High severity findings

TRST-H-1 User fee token balance can be drained in a single operation by a malicious bot

- **Category:** Gas assumptions
- **Source:** FeePayer.sol
- **Status:** Acknowledged

Description

In *_buildFeeExecutable()*, BrahRouter calculates the total fee charged to the wallet. It uses **tx.gasprice** to get the gas price specified by the bot.

```
if (feeToken == ETH) {
    uint256 totalFee = (gasUsed + GAS_OVERHEAD_NATIVE) * tx.gasprice;
    totalFee = _applyMultiplier(totalFee);
    return (totalFee, recipient,
TokenTransfer._nativeTransferExec(recipient, totalFee));
} else {
    uint256 totalFee = (gasUsed + GAS_OVERHEAD_ERC20) * tx.gasprice;
    // Convert fee amount value in fee token
    uint256 feeToCollect =

PriceFeedManager(_addressProvider.priceFeedManager()).getTokenXPriceI
nY(totalFee, ETH, feeToken);
    feeToCollect = _applyMultiplier(feeToCollect);
    return (feeToCollect, recipient,
TokenTransfer._erc20TransferExec(feeToken, recipient, feeToCollect));
}
```

The issue is that a malicious bot can manipulate **tx.gasprice** to be as high as they wish. This value is calculated post EIP1559 as the block base fee plus the sender's priority fee. A bot can offer an extremely high priority fee to drain the user's fee token balance. These losses will go to the Brahma fund manager.

Recommended mitigation

Use a gas oracle or a capped priority fee to ensure an inflated gas price down not harm the user.

Team response

There exist some scenarios where high gas may be required for quick block inclusion like liquidation protection. An additional check is not worth the added oracle gas cost for this.

We use reputable 3rd party bots like gelato which work in a decentralized fashion for bot operators. operators stake GEL tokens which get slashed if they submit txns with high gas price. Even if they do so, they have less economic incentive to do so as the gas fee will be

burned rather than being paid to the miner. If a 3rd party bot still tries to abuse it, they can be kicked by the governance using BotManager.sol.

TRST-H-2 Users can drain Gelato deposit at little cost

- **Category:** Logical flaws
- **Source:** BrahRouter.sol
- **Status:** Fixed

Description

In Console automation, fees are collected via the *claimExecutionFees()* modifier:

```
modifier claimExecutionFees(address _wallet) {
    uint256 startGas = gasleft();
    _;
    if (feeMultiplier > 0) {
        address feeToken = FeePayer._feeToken(_wallet);
        uint256 gasUsed = startGas - gasleft();
        (uint256 feeAmount, address recipient, Types.Executable
memory feeTransferTxn) =
            FeePayer._buildFeeExecutable(gasUsed, feeToken);
        emit FeeClaimed(_wallet, feeToken, feeAmount);
        if (feeToken != ETH) {
            uint256 initialBalance =
IERC20(feeToken).balanceOf(recipient);
            _executeSafeERC20Transfer(_wallet, feeTransferTxn);
            if (IERC20(feeToken).balanceOf(recipient) -
initialBalance < feeAmount) {
                revert UnsuccessfulFeeTransfer(_wallet, feeToken);
            }
        } else {
            uint256 initialBalance = recipient.balance;
            Executor._executeOnWallet(_wallet, feeTransferTxn);
            if (recipient.balance - initialBalance < feeAmount) {
                revert UnsuccessfulFeeTransfer(_wallet, feeToken);
            }
        }
    }
}
```

The actual strategy processing happens in **_;** , then the total fee is calculated using the gas used up to this point. The issue is that a malicious user can get arbitrary execution in the actual payment stage. It is easy to create and register a custom wallet contract as a Safe contract. In *_executeOnWallet()*, the contract would be called to deliver the payment request. However, the malicious contract can execute anything at this point. It could use up a large amount of gas and convert it to gas tokens like CHI. It would only pay for the strategy execution, which could be negligible compared to actual gas usage. The Gelato deposit in GelatoBot could be drained with little cost.

Recommended mitigation

When calculating fees in *buildFeeExecutable()*, there are assumptions about the gas cost of an ERC20 transfer and a native transfer.

```
// Keeper network overhead - 150k
uint256 internal constant GAS_OVERHEAD_NATIVE = 150_000 + 40_000;
uint256 internal constant GAS_OVERHEAD_ERC20 = 150_000 + 90_000;
```

A good fix would be to check the actual gas usage and require it to be under the hard cap.

Team response

Added a gas check for this attack.

Mitigation review

Applied fix has been applied.

TRST-H-3 Attackers can drain users over time by donating negligible ERC20 amount

- **Category:** Logical flaws
- **Source:** DCACoWAutomation.sol
- **Status:** Fixed

Description

In the Console automation model, a strategy shall keep executing until its trigger check fails. For DCA strategies, the swapping trigger is defined as:

```
function canInitSwap(address subAccount, address inputToken, uint256
interval, uint256 lastSwap)
    external
    view
    returns (bool)
{
    if (hasZeroBalance(subAccount, inputToken)) {
        return false;
    }
    return ((lastSwap + interval) < block.timestamp);
}
```

Note that every execution will charge the user for gas costs, which can be expensive. The issue is that whenever the account is empty, anyone can donate **inputToken** to the user, which will make the check pass periodically. It would keep executing a negligible swap order. The bot will not unsubscribe the user from the strategy because the exit trigger is defined to be zero balance for the user, which attacker can ensure will never be true.

Recommended mitigation

Define a **DUST_AMOUNT**, below that amount exit is allowed, while above that amount swap execution is allowed. User should only stand to gain from another party donating ERC20 tokens to their account.

Team response

Fix DCA automation to only swap requested amount.

Mitigation review

Fixed.

TRST-H-4 Executors can drain the Gelato deposit while profiting from free gas

- **Category:** Reentrancy flaws
- **Source:** BrahRouter.sol
- **Status:** Fixed

Description

As discussed, *claimExecutionFees()* charges the user for gas fees. Since wallet is not trusted, the payment is wrapped and checked that the balance increased by the required amount.

```
if (feeToken != ETH) {
    uint256 initialBalance = IERC20(feeToken).balanceOf(recipient);
    _executeSafeERC20Transfer(_wallet, feeTransferTxn);
    if (IERC20(feeToken).balanceOf(recipient) - initialBalance <
feeAmount) {
        revert UnsuccessfulFeeTransfer(_wallet, feeToken);
    }
} else {
    uint256 initialBalance = recipient.balance;
    Executor._executeOnWallet(_wallet, feeTransferTxn);
    if (recipient.balance - initialBalance < feeAmount) {
        revert UnsuccessfulFeeTransfer(_wallet, feeToken);
    }
}
```

While the check is sound, it can be bypassed because the execution functions are not protected from reentrancy. A bot attacker can profit by registering a malicious wallet contract, which reenters for the execution of another strategy on their wallet. At this point, the **initialBalance** will be the same as the previous **initialBalance**. The contract may reenter many times, and at the final iteration it will actually pay the **feeAmount**. When the TXs unwind, it will appear as the fee has been paid for all transactions, although it has only been paid for the last one. This can be abused to claim the gas costs from the GelatoBot deposit, while converting the free gas to gas tokens to net profit.

Recommended mitigation

Mark the execution functions as **nonReentrant**.

Team response

Adding reentrancy check.

Mitigation review

Both execution functions are protected by a reentrancy guard.

Medium severity findings

TRST-M-1 When FeePayer is subsidizing, users can steal gas

- **Category:** Logical flaws
- **Source:** FeePayer.sol
- **Status:** Acknowledged

Description

The **feeMultiplier** enables the admin to subsidize or upcharge for the automation service.

```
/**
 * @notice feeMultiplier represents the total fee to be charged on
the transaction
 *  Is set to 100% by default
 * @dev In case feeMultiplier is less than BASE_BPS, fees charged
will be less than 100%,
 *  subsidizing the transaction
 *  In case feeMultiplier is greater than BASE_BPS, fees charged will
be greater than 100%,
 *  charging the user for the transaction
 */
uint16 public feeMultiplier = 10_000;
```

The normal fee is calculated and then processed by the multiplier.

```
if (feeToken == ETH) {
    uint256 totalFee = (gasUsed + GAS_OVERHEAD_NATIVE) * tx.gasprice;
    totalFee = _applyMultiplier(totalFee);
    return (totalFee, recipient,
TokenTransfer._nativeTransferExec(recipient, totalFee));
} else {
```

The issue is that Brahma stands to lose up to **Gelato deposit x subsidization %**. Gas can be stolen by setting up a malicious wallet and programming the adapter function (likely *execTransactionFromModuleReturnData()*), to mint a popular gas token such as CHI or GST2. Then, attacker can schedule an ever-occurring automation which will trigger their gas minting logic.

Recommended mitigation

The root cause is that the **gasUsed** amount is subsidized as well as **GAS_OVERHEAD_NATIVE**, which is the gas reserved for the delivery from Gelato executors. By subsidizing only the Gelato gas portion, users will not gain from gas minting attacks, while the intention of improving user experience is maintained.

Team response

Thanks, duly noted. The subsidy is meant to be used during small-time promotional events. Acknowledged.

TRST-M-2 Strategy actions could be executed out of order due to lack of reentrancy guard

- **Category:** Reentrancy flaws
- **Source:** BrahRouter.sol
- **Status:** Fixed

Description

The Execute module performs automation of the fetched **Executable** array on wallet subaccounts.

```
function _executeAutomation(
    address _wallet,
    address _subAccount,
    address _strategy,
    Types.Executable[] memory _actionExecs
) internal {
    uint256 actionLen = _actionExecs.length;
    if (actionLen == 0) {
        revert InvalidActions();
    } else {
        uint256 idx = 0;
        do {
            _executeOnSubAccount(_wallet, _subAccount, _strategy,
_actionExecs[idx]);
            unchecked {
                ++idx;
            }
        } while (idx < actionLen);
    }
}
```

Note that there are no uses of reentrancy guards in this function or any above it in the call chain. As a result, a bot or a keeper can execute operations out of sequence. For example, instead of actions called as **[1,2,3]**, it could call **[1,[1,2,3],2,3]**. As a result, security guarantees are bypassed which can have unforeseen impacts on future strategies. Using the DCA strategy as an example, consider that it is composed of two actions. The first performs the swap while the second updates the **block.timestamp** of the last swap, so that a future automation will need to wait the appropriate interval. Through abusing the reentrancy, a bot could execute the strategy an unlimited number of times. As a result, it becomes a single spot swap instead of a DCA swap. A requirement for this attack is that at some point during execution, the strategy will interact with an external address that belongs to the keeper or bot.

Recommended mitigation

Add a reentrancy guard for *executeAutomationViaBot()* and *executeTrustedAutomation()*.

Team response

Adding reentrancy check.

Mitigation review

Fixed as suggested.

TRST-M-3 Anyone can make creating strategies extremely expensive for the user

- **Category:** Gas-bomb attacks
- **Source:** SubAccountRegistry.sol
- **Status:** Fixed

Description

In Console architecture, users can deploy spare subaccounts (Gnosis Safes) so that when they will subscribe to a strategy most of the gas spending would have been spent at a low-gas phase.

```
function deploySpareSubAccount(address _wallet) external {
    address subAccount =
SafeDeployer(addressProvider.safeDeployer()).deploySubAccount(_wallet
);
    subAccountToWalletMap[subAccount] = _wallet;
    walletToSubAccountMap[_wallet].push(subAccount);
    // No need to update subAccountStatus as it is already set to
false
    emit SubAccountAllocated(_wallet, subAccount);
}
```

The issue is that anyone can call the deploy function and specify another user's wallet. While on the surface that sounds like donating gas costs, in practice this functionality can make operating with strategies prohibitively expensive.

When users will subscribe to strategies, the StrategyRegistry will request a subaccount using this function:

```
function requestSubAccount(address _wallet) external returns
(address) {
    if (msg.sender != subscriptionRegistry) revert
OnlySubscriptionRegistryCallable();
    // Try to find a subAccount which already exists
    address[] memory subAccountList = walletToSubAccountMap[_wallet];
```

At this point, the entire subaccount array will be copied from storage to memory. Therefore, attackers can fill the array with hundreds of elements at a low-gas time and make creation of strategies very difficult.

Recommended mitigation

Limit the amount of spare subaccount to something reasonable, like 10.

Team response

Removing the spare subaccount deployment

Mitigation review

Attack surface has been removed.

TRST-M-4 DCA strategies may not be effective at averaging spot prices

- **Category:** Time-related flaws
- **Source:** CoWDCAStrategy.sol, TrustedCoWDCAStrategy.sol
- **Status:** Acknowledged

Description

The DCA strategies periodically set off CoW Swap orders, according to the specified **interval**. The order TTL is fixed at 2 hours. When **interval** is not an order of magnitude larger than **TTL**, the strategy does not average effectively. An order fulfilled near the end of the TTL window would be priced much closer to the next order. When **interval < TTL**, consecutive orders can be executed at the same time.

Recommended mitigation

TTL should be a dynamic value based on the user's specified **interval**.

Team response

User makes this DCA subscription call with interval that they decide. We don't support intervals less than a day. If they chose to try interval less than a day, which means less than effective averaging, that's their prerogative.

TRST-M-5 DCA Strategies build orders that may not be executable, wasting fees

- **Category:** Logical flaws
- **Source:** CoWDCAStrategy.sol, TrustedCoWDCAStrategy.sol
- **Status:** Fixed

Description

In *_buildInitiateSwapExecutable()*, DCA strategies determine the swap parameters for the CoW Swap. The code has recently been refactored so that there may be more than one active order simultaneously. The issue is that the function assumes the user's entire ERC20 balance to be available for the order being built.

```
// Check if enough balance present to swap, else swap entire balance
uint256 amountIn = (inputTokenBalance < params.amountToSwap) ?
inputTokenBalance : params.amountToSwap;
```

This is a problem because if the previous order will be executed before the current order, there may not be enough funds to pull from the user to execute the swap. As a result, transaction execution fees are wasted.

Recommended mitigation

Ensure only one swap can be in-flight at a time, or deduct the in-flight swap amounts from the current balance.

Team response

Set min interval during initialization to more than 2 hours.

Mitigation review

Fix works as **interval > TTL** at any point.

TRST-M-6 User will lose all Console functionality when upgrading their wallet and an upgrade target has not been set up

- **Category:** Initialization issues
- **Source:** WalletRegistry.sol
- **Status:** Fixed

Description

Console supports upgrading of the manager wallet using the *upgradeWalletType()* function.

```
function upgradeWalletType() external {
    if (!isWallet(msg.sender)) revert WalletDoesntExist(msg.sender);
    uint8 fromWalletType = _walletDataMap[msg.sender].walletType;
    _setWalletType(msg.sender, _upgradablePaths[fromWalletType]);
    emit WalletUpgraded(msg.sender, fromWalletType,
_upgradablePaths[fromWalletType]);
}
```

Note that **upgradablePaths** are set by governance. There is a lack of check that the upgradable path is defined before performing the upgrade.

```
function _setWalletType(address _wallet, uint8 _walletType) private {
    _walletDataMap[_wallet].walletType = _walletType;
}
```

If **_upgradablePaths[fromWalletType]** is zero (uninitialized), the user's wallet type shall become zero too. However, zero is an invalid value, as defined by the *isWallet*() view function:

```
function isWallet(address _wallet) public view returns (bool) {
    WalletData memory walletData = _walletDataMap[_wallet];
    if (walletData.walletType == 0 || walletData.feeToken ==
address(0)) {
        return false;
    }
    return true;
}
```

As a result, most of the functionality of Console is permanently broken when users upgrade their wallet when an upgrade path isn't set. They can salvage their funds if it is a Safe account, as they can still execute on it directly.

Recommended mitigation

When settings a new wallet type, make sure the new type is not zero.

Team response

Remove upgradable feature, replaced with deregistering wallet.

Mitigation review

Relevant code has been removed.

Low severity findings

TRST-L-1 CoW Swap orders are seen as filled although they are cancelled

- **Category:** Logical flaws
- **Source:** CoWv2Controller.sol
- **Status:** Fixed

Description

CoW Swap orders prepared by DCAStrategy can be cancelled using the following function.

```
function cancelCowOrder(bytes calldata orderUid) internal {
    settlement.setPreSignature(orderUid, false);
    settlement.invalidateOrder(orderUid);
}
```

Order status is viewed by *isOrderFullyFilled()*:

```
function isOrderFullyFilled(bytes calldata orderUid) internal view
returns (bool) {
    return settlement.filledAmount(orderUid) > 0;
}
```

An issue arises from the fact *invalidateOrder* on GPv2Settlement sets the **filledAmount** to MAX_UINT256.

```
function invalidateOrder(bytes calldata orderUid) external {
    (, address owner, ) = orderUid.extractOrderUidParams();
    require(owner == msg.sender, "GPv2: caller does not own order");
    filledAmount[orderUid] = uint256(-1);
    emit OrderInvalidated(owner, orderUid);
}
```

As a result, the order is wrongly viewed as filled.

Recommended mitigation

Remove the *invalidateOrder()* call as *setPreSignature(order_uid,false)* is adequate to cancel the order.

Team response

Adding valid cancellation order.

Mitigation review

Fix has been applied.

TRST-L-2 Rounding error causes an additional iteration of DCA strategies

- **Category:** Precision loss errors
- **Source:** CoWDCAStrategy.sol, TrustedCoWDCAStrategy.sol
- **Status:** Fixed

Description

Both CoW strategies receive an **interval** and total **amountIn** of tokens to swap. They calculate the amount per iteration as below:

```
Types.TokenRequest[] memory tokens = new Types.TokenRequest[](1);
tokens[0] = Types.TokenRequest({token: inputToken, amount:
amountIn});
amountIn = amountIn / iterations;
StrategyParams memory params = StrategyParams({
    tokenIn: inputToken,
    tokenOut: outputToken,
    amountToSwap: amountIn,
    interval: interval,
    remitToOwner: remitToOwner
});
```

There is a precision loss error in the **amountIn** calculation. It will result in an additional iteration to be executed with the division remainder, as **amountIn > amountIn / iterations * iterations**. There will be gas wasted for another automation call.

Recommended mitigation

Change the amount requested from the management wallet to **amountIn / iterations * iterations**.

Team response

Used **amountIn** as minimum order amount.

Mitigation review

Refactor in order processing makes the issue no longer possible. Money will be returned from subaccount on the last iteration.

TRST-L-3 Fee mismatch between contracts can make strategies unusable

- **Category:** Logical flaws
- **Source:** DCACoWAutomation.sol
- **Status:** Fixed

Description

In CoW Swap strategies, fee is set in the strategy contracts and then passed to *initiateSwap()*. It is built in *_buildInitiateSwapExecutable()*:

```
// Generate executable to initiate swap on DCACoWAutomation
return Types.Executable({
    callType: Types.CallType.DELEGATECALL,
    target: dcaCoWAutomation,
    value: 0,
    data: abi.encodeCall(
        DCACoWAutomation.initiateSwap,
        (params.tokenIn, params.tokenOut, swapRecipient, amountIn,
minAmountOut, swapFee)
        )
});
```

There is a mismatch between the constraints around fees between the strategy contracts and the *initiateSwap()* function:

```
function setSwapFee(uint256 _swapFee) external {
    _onlyGov();
    if (_swapFee > 10_000) {
        revert InvalidSlippage();
    }
    swapFee = _swapFee;
}
```

```
if (feeBps > 0) {
    if (feeBps > 1_000) revert FeeTooHigh();
    amountIn = amountToSwap * (MAX_BPS - feeBps) / MAX_BPS;
```

If admins set the **swapFee** to be between 1000 and 10000, it will brick all uses of the strategies.

Recommended mitigation

Enforce the same constraints on the fee percentage in both contracts, or remove the check from one of them as part of a simplified security model.

Team response

Set feeBPS to 1000 max in both.

Mitgation review

Fixed applied.

Additional recommendations

Explicit matching of enum values

In *_packMultisendTxns()*, the TX call type is checked.

```
// Enum.Operation.Call is 0
uint8 call = uint8(Enum.Operation.Call);
if (_txns[i].callType == Types.CallType.DELEGATECALL) {
    call = uint8(Enum.Operation.DelegateCall);
} else if (_txns[i].callType == Types.CallType.STATICCALL) {
    revert InvalidMultiSendCall();
}
```

When **callType** is not **DELEGATECALL** or **STATICCALL**, it assumes it is a normal call. From a design perspective, it is better to always explicitly match types, as future changes, such as adding new enum values, may silently break such assumptions and cause significant issues.

Redundant casting

In *initRegistry()*, the registry contracts are converted to addresses although they are already defined as such.

```
if (key == RegistryKey.STRATEGY) {
    _firstInit(address(strategyRegistry));
    strategyRegistry = (_newAddress);
} else if (key == RegistryKey.SUBSCRIPTION) {
    _firstInit(address(subscriptionRegistry));
    subscriptionRegistry = (_newAddress);
```

Emit events on important state changes

When the BrahRouter is paused using *setEmergencyPause()*, no event is emitted. Similarly, there is no event when the **feeMultiplier** changes. It is very recommended to emit events on important lifecycle changes for transparency.

GelatoBot creates tasks with unused _execDataOrSelector

When initiating tasks, GelatoBot uses the following code:

```
bytes32 currentTask = automate.createTask(
    address(this), // _execAddress
    abi.encodeCall(this.execute, (strategy, wallet, subAccount,
automationId)), // _execDataOrSelector
    moduleData, // _moduleData
    address(0) // _feeToken
);
```

However, since it uses the **RESOLVER** module, the **_execDataOrSelector** parameter is unused, as the actual parameter passed to the **execAddress** is what is returned from the resolver. It is recommended to pass an empty **bytes** type to save gas and improve clarity.

Missing zero-address checks in Registry contracts

The AddressProviderService inherited by the registry contracts copies the registry values internally in the constructor.

```
constructor(address _addressProvider) {
    if (_addressProvider == address(0)) revert
InvalidAddressProvider();
    addressProvider = AddressProvider(_addressProvider);
    strategyRegistry = addressProvider.strategyRegistry();
    subscriptionRegistry = addressProvider.subscriptionRegistry();
    subAccountRegistry = addressProvider.subAccountRegistry();
    walletAdapterRegistry = addressProvider.walletAdapterRegistry();
    walletRegistry = addressProvider.walletRegistry();
}
```

If one of the registries is not initialized, the constructor will exit gracefully. It is worth adding this validation which will only occur per construction and not cost much gas.

Centralization risks

CoW-Swap risks

The DCA strategy approves user funds to the **vaultRelayer** contract in the CoW infrastructure. If 3rd-party contracts are compromised or contain a vulnerability in the order processing, users could loss up to the entire order amount.

Governance-related risks

There are several ways in which a malicious or hacked governance can take user funds. Through the WalletAdapterRegistry, governance can register new wallet adapters arbitrarily. A malicious adapter can be abused easily as it can transform an incoming benign transaction to a draining TX.

```
function _executeOnWallet(address _wallet, Types.Executable memory
_txn) internal returns (bytes memory) {
    if (isEmergencyPaused) revert EmergencyPaused();
    // Get wallet adapter for the wallet
    IWalletAdapter adapter = findWalletAdapter(_wallet);
    // Format transaction into a BrahRouter compatible executable
    Types.Executable memory formattedTxn =
adapter.formatForWallet(_wallet, _txn);
    // Execute transaction
    (bool success, bytes memory result) = _execute(formattedTxn);
    // Check if transaction was successful and return data
    require(success, string(abi.encodePacked("brahRouter call fail",
result)));
    (bool txnSuccess, bytes memory txnResult) =
adapter.decodeReturnData(result);
    require(txnSuccess, string(abi.encodePacked("wallet call fail",
txnResult)));
    return (txnResult);
}
```

Governance can also call *setFeeMultiplier()* to instantly bump the fee up to 140%. This way, a user would end up paying far more than they expected for the TX.

Governance can set up a malicious price feed, which could drain user funds through the fee mechanism.

Console-admin risks

Through the trusted execution infrastructure, the delegated Console keeper calls this function for automation:

```
function executeTrustedAutomation(
    address _wallet,
```

```
    address _subAccount,
    address _strategy,
    Types.Executable[] memory _actionExecs
) external claimExecutionFees(_wallet) {
    Authorizer._validateConsoleKeeper();
    Authorizer._validateTrustedSubscription(_subAccount, _strategy);
    Executor._executeAutomation(_wallet, _subAccount, _strategy,
_actionExecs);
    emit TrustedExecution(_subAccount, _strategy);
}
```

As can be seen, it passes an Executable array, instead of requesting it from the strategy. Also, no trigger checks take place. The only check is done when invoking the action on **subaccount**:

```
function _executeOnSubAccount(address _owner, address _subAccount,
address strategy, Types.Executable memory _txn)
    internal
    returns (bytes memory)
{
    if (!IStrategy(strategy).isExecutionAllowed(_txn.callType,
_txn.target, _subAccount)) revert StrategyBlocked();
```

Strategy needs to approve the (**callType, target, subaccount**) combo. In Trusted-DCA strategy, the check is not satisfactory to protect against malicious wallet drains. It's not possible for the function to decode the recipient data as it does not receive the **calldata**.

Gelato risks

The GelatoBot contract deposits ETH for gas fees into the Gelato system. If Gelato, due to malicious reasons or due to a bug, charges more gas than required for the transaction, the balance would be deducted by this amount. It is worth noting that Gelato is planning to decentralize its Executor infrastructure, meaning anyone could potentially abuse the Gelato automation.