// Security Assessment 05.05.2025 - 05.06.2025

TokenDistributor

Agentcoin TV

HALBORN

TokenDistributor - Agentcoin TV

Prepared by: HALBORN

Last Updated 05/20/2025

Date of Engagement: May 5th, 2025 - May 6th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	0	1	1	4

TABLE OF CONTENTS

1. Introduction

Agentcoin TV engaged Halborn to conduct a security assessment on their smart contracts beginning on May 12th, 2025 and ending on May 13th, 2025. The security assessment was scoped to the smart contracts provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

The Agentcoin TV codebase in scope consists of a smart contract for complex distributions and conversions of tokens.

2. Assessment Summary

Halborn was provided 2 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Agentcoin TV team. The main ones are the following:

- Add validation for both array length and non-zero values, implement a configurable default slippage value and/or use time-weighted price data.
- Implement a role-based system for access control, transition control to a multi-signature wallet and/or establish time locks for sensitive parameter changes.
- Consider refactoring the mappings to use named arguments.
- Implement explicit validation for pool configurations before attempting to use them in swap operations and use a reversal message appropriately.

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture, purpose and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could led to arithmetic related vulnerabilities.
- Local testing with custom scripts (`Foundry`).
- Fork testing against main networks (`Foundry`).
- Static analysis of security for scoped contract, and imported functions (`Slither`).

4. Static Analysis Report

4.1 Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

4.2 Output

The findings obtained as a result of the Slither scan were reviewed, and some were not included in the report because they were determined as false positives.

```
INFO:Detectors:
TokenDistributor._encodeFunctionCall(bytes4,bytes12,address,address,uint256).argsData (src/TokenDistributor.sol#478) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
TokenDistributor._execAction(Action,ActionArgs,uint256,address,uint256[],uint256,uint256) (src/TokenDistributor.sol#389-467) ignores return value by IERC20(_paymentToken).approve(target,_amount) (src/TokenDistributor.sol#452)
TokenDistributor._execAction(Action,ActionArgs,uint256,address,uint256[],uint256,uint256) (src/TokenDistributor.sol#389-467) ignores return value by IERC20(_paymentToken).approve(target,0) (src/TokenDistributor.sol#454)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Reentrancy in TokenDistributor.distributeERC20(bytes32,address,uint256,address,address,uint256[],uint256) (src/TokenDistributor.sol#303-329):
        External calls:
        - _execDistribution(distributionId,args,_amount,_paymentToken,_minAmountsOut,0,_deadline) (src/TokenDistributor.sol#326)
                - IERC20(_tokenIn).approve(address(_permit2),type()(uint256).max) (src/libraries/UniSwapper.sol#45)
                - IBurnable(_paymentToken).burn(_amount) (src/TokenDistributor.sol#402)
                - IPermit2(_permit2).approve(_tokenIn,address(_universalRouter),uint160(_amountIn),uint48(_deadline) + 1) (src/libraries/UniSwapper.sol#48)
                - address(_recipient).sendValue(_amount) (src/TokenDistributor.sol#522)
                - (success,returndata) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#38)
                - address(_beneficiary).sendValue(_amount) (src/TokenDistributor.sol#524)
                - IWETH(_weth).withdraw(_amount) (src/TokenDistributor.sol#568)
                - IWETH(_weth).deposit{value: _amount}() (src/TokenDistributor.sol#548)
                - address(_recipient).sendValue(_amount) (src/TokenDistributor.sol#571)
                - (ok,None) = target.call{value: _amount}(callData) (src/TokenDistributor.sol#443)
                - address(_args.recipientOnFailure).sendValue(_amount) (src/TokenDistributor.sol#446)
                - IERC20(_paymentToken).approve(target,_amount) (src/TokenDistributor.sol#452)
                - (ok_scope_0,None) = target.call(callData) (src/TokenDistributor.sol#453)
                - IERC20(_paymentToken).approve(target,0) (src/TokenDistributor.sol#454)
                - _universalRouter.execute{value: _amountIn}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - _universalRouter.execute{value: 0}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - out = UniSwapper.swapExactIn(address(this),poolConfig,_paymentToken,_action.token,_amount,uint128(_minAmountsOut[_minAmountsOutIndex]),_deadline,uniswapUniversalRouter,permit2) (src/TokenDistributor.sol#417-427)
                - out = UniSwapper.swapExactIn(address(this),poolConfig,_paymentToken,_action.token,_amount,0,_deadline,uniswapUniversalRouter,permit2) (src/TokenDistributor.sol#417-427)
        External calls sending eth:
        - _execDistribution(distributionId,args,_amount,_paymentToken,_minAmountsOut,0,_deadline) (src/TokenDistributor.sol#326)
                - (success,returndata) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#38)
                - IWETH(_weth).deposit{value: _amount}() (src/TokenDistributor.sol#548)
                - (ok,None) = target.call{value: _amount}(callData) (src/TokenDistributor.sol#443)
                - _universalRouter.execute{value: _amountIn}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - _universalRouter.execute{value: 0}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
        Event emitted after the call(s):
        - Distribution(_distributionName,msg.sender,_beneficiary,_amount,_paymentToken,_recipientOnFailure,distributionId) (src/TokenDistributor.sol#328)
Reentrancy in TokenDistributor.distributeETH(bytes32,address,address,uint256[],uint256) (src/TokenDistributor.sol#268-292):
        External calls:
        - _execDistribution(distributionId,args,amount,address(0),_minAmountsOut,0,_deadline) (src/TokenDistributor.sol#289)
                - IERC20(_tokenIn).approve(address(_permit2),type()(uint256).max) (src/libraries/UniSwapper.sol#45)
                - IBurnable(_paymentToken).burn(_amount) (src/TokenDistributor.sol#402)
                - IPermit2(_permit2).approve(_tokenIn,address(_universalRouter),uint160(_amountIn),uint48(_deadline) + 1) (src/libraries/UniSwapper.sol#48)
                - address(_recipient).sendValue(_amount) (src/TokenDistributor.sol#522)
                - (success,returndata) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#38)
                - address(_beneficiary).sendValue(_amount) (src/TokenDistributor.sol#524)
                - IWETH(_weth).withdraw(_amount) (src/TokenDistributor.sol#568)
                - IWETH(_weth).deposit{value: _amount}() (src/TokenDistributor.sol#548)
                - address(_recipient).sendValue(_amount) (src/TokenDistributor.sol#571)
                - (ok,None) = target.call{value: _amount}(callData) (src/TokenDistributor.sol#443)
                - address(_args.recipientOnFailure).sendValue(_amount) (src/TokenDistributor.sol#446)
                - IERC20(_paymentToken).approve(target,_amount) (src/TokenDistributor.sol#452)
                - (ok_scope_0,None) = target.call(callData) (src/TokenDistributor.sol#453)
                - IERC20(_paymentToken).approve(target,0) (src/TokenDistributor.sol#454)
                - _universalRouter.execute{value: _amountIn}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - _universalRouter.execute{value: 0}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - out = UniSwapper.swapExactIn(address(this),poolConfig,_paymentToken,_action.token,_amount,uint128(_minAmountsOut[_minAmountsOutIndex]),_deadline,uniswapUniversalRouter,permit2) (src/TokenDistributor.sol#417-427)
                - out = UniSwapper.swapExactIn(address(this),poolConfig,_paymentToken,_action.token,_amount,0,_deadline,uniswapUniversalRouter,permit2) (src/TokenDistributor.sol#417-427)
        External calls sending eth:
        - _execDistribution(distributionId,args,amount,address(0),_minAmountsOut,0,_deadline) (src/TokenDistributor.sol#289)
                - (success,returndata) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#38)
                - IWETH(_weth).deposit{value: _amount}() (src/TokenDistributor.sol#548)
                - (ok,None) = target.call{value: _amount}(callData) (src/TokenDistributor.sol#443)
                - _universalRouter.execute{value: _amountIn}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
                - _universalRouter.execute{value: 0}(commands,inputs,_deadline) (src/libraries/UniSwapper.sol#116-120)
        Event emitted after the call(s):
        - Distribution(_distributionName,msg.sender,_beneficiary,amount,address(0),_recipientOnFailure,distributionId) (src/TokenDistributor.sol#291)
```

```
INFO:Detectors:
TokenDistributor.distributeETH(bytes32,address,address,uint256[],uint256) (src/TokenDistributor.sol#268-292) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > _deadline (src/TokenDistributor.sol#281)
TokenDistributor.distributeERC20(bytes32,address,uint256,address,address,uint256[],uint256) (src/TokenDistributor.sol#303-329) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > _deadline (src/TokenDistributor.sol#316)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
TokenDistributor._execAction(Action,ActionArgs,uint256,address,uint256[],uint256,uint256) (src/TokenDistributor.sol#389-467) has a high cyclomatic complexity (15).
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cyclomatic-complexity
INFO:Detectors:
Low level call in TokenDistributor._execAction(Action,ActionArgs,uint256,address,uint256[],uint256,uint256) (src/TokenDistributor.sol#389-467):
        - (ok,None) = target.call{value: _amount}(callData) (src/TokenDistributor.sol#443)
        - (ok_scope_0,None) = target.call(callData) (src/TokenDistributor.sol#453)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Slither:. analyzed (135 contracts with 100 detectors), 11 result(s) found
```

5. Observations

During the assessment, the `Agentcoin TV` team provided an updated version of the code (https://github.com/agentcoinorg/agent-contracts/pull/16) which integrates the `SWEEP` functionality to allow for better token handling after swap operations.

According to Uniswap documentation, the `SWEEP` functionality is recommended when:

- Dealing with native ETH operations.
- Conservative token approvals might result in excess.
- Need to ensure all tokens are properly accounted for.

This enhancement ensures that any excess native tokens resulting from swap operations are properly captured and sent to the intended recipient, preventing potential dust values from being trapped in intermediate contracts.

This improvement is noted as an observation since it represents a proactive enhancement by the `Agentcoin TV` team. Given the assessment's primary focus on the `TokenDistributor` contract functionality, and considering this update affects primarily a supporting library component, it represents a complementary improvement to the core protocol architecture that enhances the overall robustness of the system.

6. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

6.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

6.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

6.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

7. SCOPE

FILES AND REPOSITORY

(a) Repository: agent-contracts

(b) Assessed Commit ID: 26030e1

(c) Items in scope:

- src/TokenDistributor.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

Out-of-Scope: New features/implementations after the remediation commit IDs.

8. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	1

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INSUFFICIENT SLIPPAGE PROTECTION	MEDIUM	SOLVED - 05/19/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION RISKS	LOW	RISK ACCEPTED - 05/19/2025
LACK OF NAMED MAPPINGS	INFORMATIONAL	SOLVED - 05/19/2025
LACK OF EXPLICIT POOL CONFIG VERIFICATION IN SWAP LOGIC	INFORMATIONAL	SOLVED - 05/19/2025
MISSING VARIABLE VISIBILITY	INFORMATIONAL	SOLVED - 05/19/2025
USE OF MAGIC NUMBERS	INFORMATIONAL	SOLVED - 05/19/2025

9. FINDINGS & TECH DETAILS

9.1 INSUFFICIENT SLIPPAGE PROTECTION

// MEDIUM

Description

The `TokenDistributor` contract does not validate that the `_minAmountsOut` array has sufficient entries to cover all swaps in a distribution. When the array is shorter than needed, the code defaults to using `0` as the minimum output amount for any remaining swaps. Additionally, there is no validation of individual `_minAmountsOut` values, allowing users to explicitly provide zero values.

When executing swaps as part of a distribution, the `_execAction` function uses the following logic to determine minimum output amounts:

```
out = UniSwapper.swapExactIn(
    address(this),
    poolConfig,
    _paymentToken,
    _action.token,
    _amount,
    _minAmountsOutIndex < _minAmountsOut.length ? uint128(_minAmountsOut[_minAmountsOutIndex]) : 0,
    _deadline,
    uniswapUniversalRouter,
    permit2
);
```

When the `_minAmountsOutIndex` exceeds the length of the `_minAmountsOut` array, the code silently defaults to using `0` as the minimum output amount. This creates transactions that are vulnerable to sandwich attacks and other forms of MEV extraction.

The issue can be exacerbated in complex nested distributions, where calculating the exact number of required minimum amounts becomes challenging for users and integrators.

This vulnerability can lead to:

1. **Direct value extraction** through MEV attacks like sandwiching.
2. **Silent loss of funds,** since the contract continues execution without errors when slippage protection is missing.
3. **Unpredictable financial outcomes** for users of complex distributions.
4. **Security risks for protocol integrators** who rely on this contract for token distributions.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (6.7)

Recommendation

Consider the following safeguards:

1. **Add validation for both array length and non-zero values** - Prevent execution with either missing or explicitly zero slippage protection.
2. **Implement a configurable default slippage value** - Replace zeros (whether explicit or from array bounds issues) with a protocol-governed slippage percentage.
3. **Use time-weighted price data** - For critical operations, supplement with TWAP from oracles to establish reliable price boundaries.

Remediation Comment

SOLVED: The **AgentCoin TV team** solved this finding in commit `dd85680` by following the mentioned recommendation, explicitly checking for zero or missing minimum amount values, preventing transactions without proper slippage protection from executing.

Remediation Hash

https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

References

agentcoinorg/agent-contracts/src/TokenDistributor.sol#L524-L526

9.2 CENTRALIZATION RISKS

// LOW

Description

The `TokenDistributor` contract grants the owner significant control over critical protocol parameters and functionality. The owner's privileges include:

- Setting pool configurations which directly affects swap execution and token rates.
- Mapping distribution names to distribution IDs which defines protocol behavior.
- Upgrading the contract implementation.
- Adding distributions that can include `SendAndCall` actions to any external contract.

These privileges arguably create a centralization risk that could lead to owner privileges being abused if the owner becomes malicious or their credentials are compromised, potentially resulting in unfavorable swap rates, unexpected distribution behavior, or even redirection of user tokens during execution.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:C/Y:N (2.5)

Recommendation

Consider implementing several remedial strategies to reduce centralization risks, including but not limited to:

1. Implementing a role-based system for access control with separated responsibilities.
2. Transitioning control to a multi-signature wallet setup for critical functions.
3. Establishing time locks for sensitive parameter changes and contract upgrades.
4. Creating a formal governance process for approving external contracts that can be called via distributions.
5. Clearly documenting trust assumptions, so users understand the security boundaries of the protocol.

By adopting these measures, the protocol can maintain necessary administrative flexibility while providing stronger security guarantees and greater transparency to users.

Remediation Comment

RISK ACCEPTED: The **AgentCoin TV team** made a business decision to accept the risk of this finding and not alter the contracts.

References

agentcoinorg/agent-contracts/src/TokenDistributor.sol

9.3 LACK OF NAMED MAPPINGS

// INFORMATIONAL

Description

The project contains several unnamed mappings, despite using a Solidity version that supports named mappings.

Named mappings improve code readability and self-documentation by explicitly stating their purpose.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:L/D:N/Y:N (0.6)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit. For example:

```
mapping(address myAddress => bool myBool) public myMapping;
```

Remediation Comment

SOLVED: The **AgentCoin TV team** solved this finding in commit `dd85680` by following the mentioned recommendation.

Remediation Hash

https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

References

agentcoinorg/agent-contracts/src/TokenDistributor.sol#L113-L116

9.4 LACK OF EXPLICIT POOL CONFIG VERIFICATION IN SWAP LOGIC

// INFORMATIONAL

Description

The `TokenDistributor` contract does not explicitly validate the existence of pool configurations before attempting to use them in swap operations. When a swap is executed with a token pair that lacks a registered pool configuration, the transaction fails without a specific error message indicating the root cause.

```
PoolConfig memory poolConfig = pools[_getSwapPairKey(_paymentToken, _action.token)];
// No validation that poolConfig exists or is properly initialized
```

When users attempt to execute a distribution with non-configured token pairs, the transaction reverts with a generic error rather than a clear explanation. This creates a poor user experience and may lead to confusion about why their transaction failed.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:L/D:N/Y:N (0.6)

Recommendation

Implement explicit validation for pool configurations before attempting to use them in swap operations, and use a reversal message appropriately.

Remediation Comment

SOLVED: The **AgentCoin TV team** solved this finding in commit `dd85680` by following the mentioned recommendation.

Remediation Hash

https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

9.5 MISSING VARIABLE VISIBILITY

// INFORMATIONAL

Description

The visibility of the `poolProposals` variable is not explicitly defined. By default, variables are set with the `internal` visibility. However, it is considered best practice to explicitly specify visibility to enhance clarity and prevent ambiguity. Clearly labeling the visibility of all variables and functions will help in maintaining clear and understandable code.

BVSS

AO:A/AC:L/AX:H/R:F/S:U/C:N/A:N/I:L/D:N/Y:N (0.2)

Recommendation

Explicitly define the visibility of all variables in the contracts to enhance readability and reduce the potential for errors.

Remediation Comment

SOLVED: The **AgentCoin TV team** solved this finding in commit `dd85680` by following the mentioned recommendation.

Remediation Hash

https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

References

agentcoinorg/agent-contracts/src/TokenDistributor.sol#L114

9.6 USE OF MAGIC NUMBERS

// INFORMATIONAL

Description

In the `TokenDistributor` contract in scope, there are instances where magic numbers are used. Magic numbers are direct numerical or string values used in the code without any explanation or context. This practice can lead to code maintainability issues, potential errors, and difficulty in understanding the code's logic.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. Consider adding an inline comment explaining how the magic numbers are calculated or why they are chosen for complex values.

Remediation Comment

SOLVED: The **AgentCoin TV team** solved this finding in commit `dd85680` by following the mentioned recommendation.

Remediation Hash

https://github.com/agentcoinorg/agent-contracts/pull/17/commits/dd85680cd8c019e344d4f7d8c93d5df7ebc3bebc

References

agentcoinorg/agent-contracts/src/TokenDistributor.sol#L179
agentcoinorg/agent-contracts/src/TokenDistributor.sol#L464

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.