// Security Assessment

03.18.2025 - 03.20.2025

Agent Contracts

Agentcoin TV

HALBORN

Agent Contracts - Agentcoin TV

Prepared by: HALBORN

Last Updated 04/01/2025

Date of Engagement: March 18th, 2025 - March 20th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	0	0	2	3	4

TABLE OF CONTENTS

1. Introduction

Agentcoin TV engaged Halborn to conduct a security assessment on smart contracts beginning on March 18th, 2025 and ending on March 20th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn dedicated 3 days for the engagement and assigned one full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

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 were the following:

- Modify the existing launch() function to properly handle the immediate launch condition when maximum deposits are reached.
- Always check the return value of call sending etc and revert if the transfer fails.
- Add explicit zero address checks for all critical address parameters.
- Add a timelock for sensitive operations, especially contract upgrades.
- Implement a two-step ownership transfer pattern.
- Implement checks-effects-interactions pattern by properly ordering operations.
- Several gas optimizations.
- Add complete NatSpec documentation for all public and external functions.

3. Test Approach And Methodology

Halborn performed a combination of manual, semi-automated and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding 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 the code 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 and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any vulnerability classes
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Local deployment and testing (`Foundry`)

4. 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.

4.1 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:

$$E = \prod m_e$$

4.2 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:

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

4.3 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

5. SCOPE

FILES AND REPOSITORY

(a) Repository: agent-contracts

(b) Assessed Commit ID: e9e8ab7

(c) Items in scope:

- src/AgentFactory.sol
- src/AgentLaunchPool.sol
- src/AgentUniswapHook.sol
- src/AgentUniswapHookDeployer.sol
- src/BaseHookUpgradeable.sol
- src/DistributionAndPriceChecker.sol
- src/UniswapPoolDeployer.sol
- agentcoinorg/agent-contracts/pull/9
- agentcoinorg/agent-contracts/pull/10

Out-of-Scope: src/AgentToken.sol, src/AgentStaking.sol, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- a326429
- 3f6cf1b
- 7bd8b7e

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	2	3

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
IMMEDIATE LAUNCH CONDITION DISCREPANCY ON MAXIMUM DEPOSIT	MEDIUM	SOLVED - 03/21/2025
UNCHECKED EXTERNAL CALL RESULTS	MEDIUM	SOLVED - 03/25/2025
REENTRANCY ATTACK CAUSING TOKEN BALANCE MANIPULATION	LOW	SOLVED - 03/27/2025
SINGLE-STEP OWNERSHIP TRANSFER RISK	LOW	SOLVED - 03/25/2025
PUSH VS PULL PATTERN FOR COLLATERAL DISTRIBUTION	LOW	RISK ACCEPTED - 03/28/2025
MISSING ZERO ADDRESS VALIDATION	INFORMATIONAL	SOLVED - 03/25/2025
CENTRALIZATION RISKS	INFORMATIONAL	ACKNOWLEDGED - 03/28/2025
GAS OPTIMIZATION OPPORTUNITIES	INFORMATIONAL	SOLVED - 03/25/2025
MISSING NATSPEC DOCUMENTATION	INFORMATIONAL	SOLVED - 03/25/2025

7. FINDINGS & TECH DETAILS

7.1 IMMEDIATE LAUNCH CONDITION DISCREPANCY ON MAXIMUM DEPOSIT

// MEDIUM

Description

An inconsistency exists between the documented behavior and actual implementation of the launch condition in the `AgentLaunchPool` contract. According to the documentation, "if deposits reach the maximum, the launch happens right away before the time window is ended." However, the current implementation does not execute this immediate launch functionality.

In the `launch()` function, the code enforces the time window check in all scenarios:

```
function launch() external virtual {
    if (hasLaunched) {
        revert AlreadyLaunched();
    }

    if (!_hasTimeWindowPassed()) {
        revert TimeWindowNotPassed();
    }

    if (totalDeposited < launchPoolInfo.minAmountForLaunch) {
        revert MinAmountNotReached();
    }

    // Launch logic...
}
```

Impact:
1. Reduced capital efficiency as user funds remain locked for the full time window even when launch success is guaranteed
2. User confusion or potential loss of trust when documented behavior doesn't match actual contract behavior
3. Potential opportunity costs for users unable to access or trade their tokens until the entire time window elapses

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:L/D:N/Y:N (5.6)

Recommendation

Modify the existing `launch()` function to properly handle the immediate launch condition when maximum deposits are reached:

```
function launch() external virtual {
    if (hasLaunched) {
        revert AlreadyLaunched();
    }

    // Allow immediate launch if max deposit reached, otherwise enforce time window
```

```
    if (totalDeposited < launchPoolInfo.maxAmountForLaunch && !_hasTimeWindowPassed()) {
        revert TimeWindowNotPassed();
    }

    if (totalDeposited < launchPoolInfo.minAmountForLaunch) {
        revert MinAmountNotReached();
    }

    // Launch logic...
}
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

a326429bd443dccfb05f3bcea6e79f59a9626364

7.2 UNCHECKED EXTERNAL CALL RESULTS

// MEDIUM

Description

The following contract functions don't check the return value of the low-level `call` function when sending ETH:

- `AgentLaunchPool::_distributeCollateral`
- `AgentLaunchPool::reclaimETHDepositsFor`

```
_beneficiary.call{value: amount}("");  // Return value not checked

payable(launchPoolInfo.collateralRecipients[i]).call{value: amount}("");  // Return value not checked
```

If the transfer fails (e.g., if the recipient contract reverts in its receive/fallback function), the function will continue execution as if the transfer succeeded, potentially leading to accounting inconsistencies.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:H/Y:N (5.4)

Recommendation

Always check the return value of call and revert if the transfer fails:

```
(bool success, ) = _beneficiary.call{value: amount}("");
require(success, "ETH transfer failed");
```

Alternatively, use OpenZeppelin's `Address.sendValue()` which handles this check internally.

Remediation Comment

SOLVED: OpenZeppelin's `Address.sendValue()` was implemented to check for return value of external calls.

Remediation Hash

3f6cf1b9a0043aef8464ebfd753f1fa22cc44076

7.3 REENTRANCY ATTACK CAUSING TOKEN BALANCE MANIPULATION

// LOW

Description

The `_distributeCollateral()` function enables a sophisticated reentrancy attack that manipulates token balances to cause launch failure. The attack works as follows:

1. During launch, ETH is sent to recipients using unchecked call operations
2. A malicious recipient can reenter the contract when receiving ETH
3. Since hasLaunched is already set to true, the attacker can call `claim()` to withdraw their agent tokens
4. When execution returns to `_setupInitialLiquidity()`, this check will fail:

```
if (tokenBalance < launchPoolAmount + uniswapPoolAmount) {
    revert NotEnoughTokensToDeploy();
}
```

This causes the entire launch transaction to revert.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:M/D:N/Y:N (2.3)

Recommendation

Implement checks-effects-interactions pattern by properly ordering operations:

```
function launch() external nonReentrant {
    // Validate conditions
    // Deploy tokens
    // Setup liquidity
    // Mark as launched (hasLaunched = true) - only after all critical operations
    // Distribute collateral
}
```

Remediation Comment

SOLVED: `hasDeployed` is set to `true` after launching the pool.

```
_deployAgentStaking(contractOwner, agentTokenAddress);

_setupInitialLiquidity(launchPoolInfo.collateral, agentTokenAddress);

hasLaunched = true;

_distributeCollateral();
```

Remediation Hash

7bd8b7e64e1a5a57053ccd4c32db9fb6693f57d7

7.4 SINGLE-STEP OWNERSHIP TRANSFER RISK

// LOW

Description

The following contracts use the standard OwnableUpgradeable pattern, which performs ownership transfer in a single step:

- `AgentLaunchPool::transferOwnership`
- `AgentUniswapHook::transferOwnership`

This creates a significant risk if ownership is accidentally transferred to an incorrect or inaccessible address, as there would be no way to recover control of the contract:

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:M/D:N/Y:N (2.3)

Recommendation

Implement a two-step ownership transfer pattern (like Ownable2Step from OpenZeppelin) where:
1. The current owner proposes a transfer to a new address
2. The new owner must accept the ownership in a separate transaction

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

3f6cf1b9a0043aef8464ebfd753f1fa22cc44076

7.5 PUSH VS PULL PATTERN FOR COLLATERAL DISTRIBUTION

// LOW

Description

The `AgentLaunchPool` contract uses a push pattern for distributing collateral, where the contract actively sends ETH/tokens to recipients during the distribution process.

```
    function _distributeCollateral() internal virtual {
        uint256 length = launchPoolInfo.collateralRecipients.length;

        uint256 cachedTotalDeposited = totalDeposited;

        if (launchPoolInfo.collateral == address(0)) {
            for (uint256 i = 0; i < length; ++i) {
                uint256 amount = launchPoolInfo.collateralBasisAmounts[i] * cachedTotalDeposited / 1e4;
                payable(launchPoolInfo.collateralRecipients[i]).sendValue(amount);
            }
        } else {
            for (uint256 i = 0; i < length; ++i) {
                uint256 amount = launchPoolInfo.collateralBasisAmounts[i] * cachedTotalDeposited / 1e4;
                IERC20(launchPoolInfo.collateral).safeTransfer(launchPoolInfo.collateralRecipients[i], amount);
            }
        }
    }
```

This approach has several drawbacks:

1. **Gas Costs**: The contract pays gas for each transfer, which can be expensive when there are many recipients
2. **Failed Transfers**: If a recipient is a contract that reverts in its receive/fallback function, the entire distribution process fails
3. **No Retry Mechanism**: Failed transfers require manual intervention to retry
4. **Block Gas Limits**: Large distributions might hit block gas limits

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (2.1)

Recommendation

Implement a pull pattern where recipients can withdraw their funds themselves:

```
// Tracking recipient balances
mapping(address => uint256) public pendingCollateralWithdrawals;

// Function to distribute collateral (sets up the amounts for withdrawal)
function _distributeCollateral() internal virtual {
    //...

    pendingCollateralWithdrawals[recipient] += amount;

    //...
}

// Function for recipients to withdraw their collateral
function withdrawCollateral() external {
   //...
}
```

Remediation Comment

RISK ACCEPTED: The **Agentcoin team** accepted the risk related to this finding

7.6 MISSING ZERO ADDRESS VALIDATION

// INFORMATIONAL

Description

Functions in the following contracts don't validate that address parameters are not the zero address (0x0):

- `AgentLaunchPool::initialize` (missing validation for `_owner` and other address parameters)
- `AgentUniswapHook::initialize` (missing validation for `_owner`, `_controller`, etc.)

If these addresses are set to the zero address (either accidentally or maliciously), it could lead to funds being lost or critical functionality becoming unusable.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:N/Y:N (1.9)

Recommendation

Add explicit zero address checks for all critical address parameters:

```
function initialize(...) external initializer {
    require(_owner != address(0), "Zero address not allowed");
    require(_tokenInfo.tokenImplementation != address(0), "Zero implementation address");
    // ... other address checks

    __Ownable_init(_owner);
    // ... rest of the initialization
}
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

3f6cf1b9a0043aef8464ebfd753f1fa22cc44076

7.7 CENTRALIZATION RISKS

// INFORMATIONAL

Description

The following contracts use the UUPS upgrade pattern with a single owner, creating centralization risks:

- `AgentLaunchPool::_authorizeUpgrade`
- `AgentUniswapHook::_authorizeUpgrade`
- Any other contract inheriting OwnableUpgradeable with upgrade capability

If the owner's private key is compromised, an attacker could upgrade the contracts to malicious implementations, potentially stealing all funds or manipulating contract behavior.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (1.5)

Recommendation

- Implement a multi-signature scheme for the owner role.
- Add a timelock for sensitive operations, especially contract upgrades.

Remediation Comment

ACKNOWLEDGED: The **Agentcoin team** acknowledged the centralization risks

7.8 GAS OPTIMIZATION OPPORTUNITIES

// INFORMATIONAL

Description

The following contract functions could benefit from gas optimizations, particularly in loops:

- `AgentLaunchPool::_distributeCollateral`
- `AgentLaunchPool::multiClaim`
- `AgentUniswapHook::takeFees`

BVSS

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

Recommendation

Several gas optimizations could be implemented:
1. Use `++i` instead of `i++` in loops (pre-increment is cheaper)
2. Cache array lengths instead of accessing them in each loop iteration
3. Use calldata instead of memory for read-only function parameters
4. Consolidate multiple storage reads/writes to the same variable

Remediation Comment

SOLVED: The suggested gas optimizations were applied.

Remediation Hash

3f6cf1b9a0043aef8464ebfd753f1fa22cc44076

7.9 MISSING NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

he following functions lack comprehensive NatSpec documentation:

- Multiple functions across `AgentLaunchPool.sol`
- Multiple functions across `AgentUniswapHook.sol`
- Functions in other contracts

Missing documentation includes:

- Missing param descriptions
- Missing return value descriptions
- Missing explanations for complex 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

Add complete NatSpec documentation for all public and external functions:

```
/// @notice Calculates the starting price for the Uniswap pool
/// @dev Uses the square root of the token ratio to determine initial price
/// @param _collateral The address of the collateral token
/// @param _agentToken The address of the agent token
/// @param _collateralAmount The amount of collateral to be used for liquidity
/// @param _agentAmount The amount of agent tokens to be used for liquidity
/// @return The calculated square root price as a uint160 encoded Q64.96 value
function _calculateUniswapStartingPrice(
    address _collateral,
    address _agentToken,
    uint256 _collateralAmount,
    uint256 _agentAmount
) internal pure virtual returns (uint160) {
    // Function implementation
}
```

Remediation Comment

SOLVED: NatSpec was added where necessary.

Remediation Hash

3f6cf1b9a0043aef8464ebfd753f1fa22cc44076

8. AUTOMATED TESTING

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 conducted a comprehensive review of findings generated by the Slither static analysis tool. No major issues were found for contracts in-scope. The vulnerabilities related to reentrancies are a false positives as call is made to a trusted contract.

```
INFO:Detectors:
AgentLaunchPool.reclaimETHDepositsFor(address) (src/AgentLaunchPool.sol#256-281) sends eth to arbitrary user
        Dangerous calls:
        - _beneficiary.call{value: amount}() (src/AgentLaunchPool.sol#276)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
```

```
INFO:Detectors:
Reentrancy in AgentLaunchPool._deployAgentStaking(address,address) (src/AgentLaunchPool.sol#407-419):
        External calls:
        - proxy = new ERC1967Proxy(address(tokenInfo.stakingImplementation),abi.encodeCall(IAgentStaking.initialize,(_owner,_agentTokenAddress))) (src/AgentLaunchPool.sol#413-416)
        State variables written after the call(s):
        - agentStaking = address(proxy) (src/AgentLaunchPool.sol#418)
        AgentLaunchPool.agentStaking (src/AgentLaunchPool.sol#74) can be used in cross function reentrancies:
        - AgentLaunchPool._deployAgentStaking(address,address) (src/AgentLaunchPool.sol#407-419)
        - AgentLaunchPool.agentStaking (src/AgentLaunchPool.sol#74)
        - AgentLaunchPool.launch() (src/AgentLaunchPool.sol#126-153)
Reentrancy in AgentLaunchPool._deployAgentToken(address) (src/AgentLaunchPool.sol#389-402):
        External calls:
        - proxy = new ERC1967Proxy(tokenInfo.tokenImplementation,tokenCtorArgs) (src/AgentLaunchPool.sol#397)
        State variables written after the call(s):
        - agentToken = address(proxy) (src/AgentLaunchPool.sol#399)
        AgentLaunchPool.agentToken (src/AgentLaunchPool.sol#73) can be used in cross function reentrancies:
        - AgentLaunchPool._claim(address) (src/AgentLaunchPool.sol#364-384)
        - AgentLaunchPool._deployAgentToken(address) (src/AgentLaunchPool.sol#389-402)
        - AgentLaunchPool.agentToken (src/AgentLaunchPool.sol#73)
        - AgentLaunchPool.computeAgentTokenAddress() (src/AgentLaunchPool.sol#157-166)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

```
INFO:Detectors:
AgentLaunchPool.depositETHFor(address) (src/AgentLaunchPool.sol#196-221) ignores return value by address(_beneficiary).call{value: msg.value - depositAmount}() (src/AgentLaunchPool.sol#217)
AgentLaunchPool.reclaimETHDepositsFor(address) (src/AgentLaunchPool.sol#256-281) ignores return value by _beneficiary.call{value: amount}() (src/AgentLaunchPool.sol#276)
AgentLaunchPool._distributeCollateral() (src/AgentLaunchPool.sol#466-480) ignores return value by address(launchPoolInfo.collateralRecipients[i]).call{value: amount}() (src/AgentLaunchPool.sol#472)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-low-level-calls
```

```
INFO:Detectors:
Reentrancy in AgentLaunchPool.depositETHFor(address) (src/AgentLaunchPool.sol#196-221):
        External calls:
        - address(_beneficiary).call{value: msg.value - depositAmount}() (src/AgentLaunchPool.sol#217)
        Event emitted after the call(s):
        - Deposit(_beneficiary,msg.sender,depositAmount) (src/AgentLaunchPool.sol#220)
Reentrancy in AgentLaunchPool.launch() (src/AgentLaunchPool.sol#126-153):
        External calls:
        - agentTokenAddress = _deployAgentToken(contractOwner) (src/AgentLaunchPool.sol#144)
                - proxy = new ERC1967Proxy(tokenInfo.tokenImplementation,tokenCtorArgs) (src/AgentLaunchPool.sol#397)
        - _deployAgentStaking(contractOwner,agentTokenAddress) (src/AgentLaunchPool.sol#146)
                - proxy = new ERC1967Proxy(address(tokenInfo.stakingImplementation),abi.encodeCall(IAgentStaking.initialize,(_owner,_agentTokenAddress))) (src/AgentLaunchPool.sol#413-416)
        - _distributeCollateral() (src/AgentLaunchPool.sol#148)
                - address(launchPoolInfo.collateralRecipients[i]).call{value: amount}() (src/AgentLaunchPool.sol#472)
        - _setupInitialLiquidity(launchPoolInfo.collateral,agentTokenAddress) (src/AgentLaunchPool.sol#150)
                - _poolInfo.poolManager.initialize(poolKey,_poolInfo.startingPrice) (src/UniswapPoolDeployer.sol#69)
                - IERC20(_poolInfo.collateral).approve(address(_poolInfo.permit2),type()(uint256).max) (src/UniswapPoolDeployer.sol#98)
                - IAllowanceTransfer(_poolInfo.permit2).approve(_poolInfo.collateral,address(_poolInfo.positionManager),type()(uint160).max,type()(uint48).max) (src/UniswapPoolDeployer.sol#99-104)
                - IERC20(_poolInfo.agentToken).approve(address(_poolInfo.permit2),type()(uint256).max) (src/UniswapPoolDeployer.sol#107)
                - IAllowanceTransfer(_poolInfo.permit2).approve(_poolInfo.agentToken,address(_poolInfo.positionManager),type()(uint160).max,type()(uint48).max) (src/UniswapPoolDeployer.sol#108-113)
                - _poolInfo.positionManager.modifyLiquidities{value: deploymentInfo.amount0Max}(abi.encode(actions,mintParams),block.timestamp) (src/UniswapPoolDeployer.sol#116)
                - _poolInfo.positionManager.modifyLiquidities(abi.encode(actions,mintParams),block.timestamp) (src/UniswapPoolDeployer.sol#118)
        External calls sending eth:
        - _distributeCollateral() (src/AgentLaunchPool.sol#148)
                - address(launchPoolInfo.collateralRecipients[i]).call{value: amount}() (src/AgentLaunchPool.sol#472)
        - _setupInitialLiquidity(launchPoolInfo.collateral,agentTokenAddress) (src/AgentLaunchPool.sol#150)
                - _poolInfo.positionManager.modifyLiquidities{value: deploymentInfo.amount0Max}(abi.encode(actions,mintParams),block.timestamp) (src/UniswapPoolDeployer.sol#116)
        Event emitted after the call(s):
        - Launch(agentTokenAddress,agentStaking) (src/AgentLaunchPool.sol#152)
Reentrancy in AgentLaunchPool.reclaimETHDepositsFor(address) (src/AgentLaunchPool.sol#256-281):
        External calls:
        - _beneficiary.call{value: amount}() (src/AgentLaunchPool.sol#276)
        Event emitted after the call(s):
        - ReclaimDeposits(_beneficiary,msg.sender,amount) (src/AgentLaunchPool.sol#278)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

```
INFO:Detectors:
AgentUniswapHook.initialize(address,address,address)._controller (src/AgentUniswapHook.sol#47) lacks a zero-check on :
              - controller = _controller (src/AgentUniswapHook.sol#54)
AgentUniswapHook.setController(address)._controller (src/AgentUniswapHook.sol#61) lacks a zero-check on :
              - controller = _controller (src/AgentUniswapHook.sol#62)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
```

```
INFO:Detectors:
BaseHookUpgradeable._afterAddLiquidity(address,PoolKey,IPoolManager.ModifyLiquidityParams,BalanceDelta,BalanceDelta,bytes) (src/BaseHookUpgradeable.sol#107-116) is never used and should be removed
BaseHookUpgradeable._afterDonate(address,PoolKey,uint256,uint256,bytes) (src/BaseHookUpgradeable.sol#208-214) is never used and should be removed
BaseHookUpgradeable._afterInitialize(address,PoolKey,uint160,int24) (src/BaseHookUpgradeable.sol#54-56) is never used and should be removed
BaseHookUpgradeable._afterRemoveLiquidity(address,PoolKey,IPoolManager.ModifyLiquidityParams,BalanceDelta,BalanceDelta,bytes) (src/BaseHookUpgradeable.sol#130-139) is never used and should be removed
BaseHookUpgradeable._afterSwap(address,PoolKey,IPoolManager.SwapParams,BalanceDelta,bytes) (src/BaseHookUpgradeable.sol#170-176) is never used and should be removed
BaseHookUpgradeable._beforeAddLiquidity(address,PoolKey,IPoolManager.ModifyLiquidityParams,bytes) (src/BaseHookUpgradeable.sol#68-74) is never used and should be removed
BaseHookUpgradeable._beforeDonate(address,PoolKey,uint256,uint256,bytes) (src/BaseHookUpgradeable.sol#189-195) is never used and should be removed
BaseHookUpgradeable._beforeInitialize(address,PoolKey,uint160) (src/BaseHookUpgradeable.sol#40-42) is never used and should be removed
BaseHookUpgradeable._beforeRemoveLiquidity(address,PoolKey,IPoolManager.ModifyLiquidityParams,bytes) (src/BaseHookUpgradeable.sol#86-93) is never used and should be removed
BaseHookUpgradeable._beforeSwap(address,PoolKey,IPoolManager.SwapParams,bytes) (src/BaseHookUpgradeable.sol#151-157) is never used and should be removed
ContextUpgradeable.__Context_init() (node_modules/@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol#18-19) is never used and should be removed
ContextUpgradeable.__Context_init_unchained() (node_modules/@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol#21-22) is never used and should be removed
ContextUpgradeable._contextSuffixLength() (node_modules/@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol#31-33) is never used and should be removed
ContextUpgradeable._msgData() (node_modules/@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol#27-29) is never used and should be removed
Initializable._getInitializedVersion() (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol#208-210) is never used and should be removed
UUPSUpgradeable.__UUPSUpgradeable_init_unchained() (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol#68-69) is never used and should be removed
add(BalanceDelta,BalanceDelta) (node_modules/@uniswap/v4-core/src/types/BalanceDelta.sol#20-32) is never used and should be removed
eq(BalanceDelta,BalanceDelta) (node_modules/@uniswap/v4-core/src/types/BalanceDelta.sol#48-50) is never used and should be removed
greaterThan(Currency,Currency) (node_modules/@uniswap/v4-core/src/types/Currency.sol#16-18) is never used and should be removed
greaterThanOrEqualTo(Currency,Currency) (node_modules/@uniswap/v4-core/src/types/Currency.sol#24-26) is never used and should be removed
lessThan(Currency,Currency) (node_modules/@uniswap/v4-core/src/types/Currency.sol#20-22) is never used and should be removed
neq(BalanceDelta,BalanceDelta) (node_modules/@uniswap/v4-core/src/types/BalanceDelta.sol#52-54) is never used and should be removed
sub(BalanceDelta,BalanceDelta) (node_modules/@uniswap/v4-core/src/types/BalanceDelta.sol#34-46) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
```

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.