Cyberscope

Audit Report

Hodl

November 2024

SHA256

57c13e1b4fddbccbb6621ff202e84ab93edce5c0d324529a866dbc53a4daaeae

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IRC	Incaccurate Reward Calculation	Unresolved
●	MAC	Missing Access Control	Unresolved
●	RCC	Redundant Calculations Complexity	Unresolved
●	URA	Unrestricted Redeemed Amount	Unresolved
●	PSAI	Potential Swap Amount Inconsistency	Unresolved
●	RCS	Redundant Conditional Statements	Unresolved
●	UAR	Unexcluded Address Restrictions	Unresolved
●	UOSA	Usage of Solidity Assembly	Unresolved
●	DKO	Delete Keyword Optimization	Unresolved
●	MMN	Misleading Method Naming	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved

●	POSD	Potential Oracle Stale Data	Unresolved
●	TUU	Time Units Usage	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Audit Updates

Initial Audit	29 Nov 2024
Test Deploy	https://sepolia.etherscan.io/address/0x934c713FdCB13435eB44FD40A0D60BC6C8d681e2

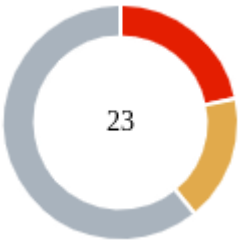
Source Files

Filename	SHA256
OwnableUpgradeable.sol	ebf38dc17b401ac3a98de2db8c31e184b52dac2c7e8ac5e53884298a8f815a0c
IHODL.sol	67634ec775564454928c13c15532201321f9c812ca130738c4cc02edd731ad7d
HODL.sol	57c13e1b4fddbccbb6621ff202e84ab93edce5c0d324529a866dbc53a4daaeae

Contract Readability Comment

The assessment of the smart contract has revealed a deeply concerning issue – the codebase is overly complicated, tangled, and deviates significantly from fundamental coding principles. The complexity has reached a level where the code becomes almost unreadable and unintelligible. Even if the identified findings are addressed and rectified, the contract would still remain far from being production-ready due to its convoluted and non-standard structure. This inherent complexity not only hampers the contract's security but also presents a considerable maintenance challenge. To ensure the contract's stability, security, and long-term viability, it is essential to conduct a comprehensive code refactor. Simplifying and restructuring the code to adhere to best practices and coding standards will be imperative for making the contract production-ready and maintainable.

Findings Breakdown



Critical	5
Medium	4
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	5	0	0	0
Medium	4	0	0	0
Minor / Informative	14	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	HODL.sol#L169,619
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the `buySellCooldown` to a high value. In addition the contract owner can set a maximum sell amount equal to 0.01% of the total supply. As a result, the contract may operate as a honeypot.

```
function _update(address from, address to, uint256 value) internal
virtual override {
  ...
  if (isPairAddress[to] && from != address(this) &&
!_isOwner(from)) {
  if (block.timestamp <= _userLastBuy[from] + buySellCooldown)
revert CooldownInEffect();
  ...

  }
  ...
}
```

```
function changeBuySellCooldown(uint256 newValue) external onlyOwner
onlyPermitted {
    uint256 oldValue = buySellCooldown;
    buySellCooldown = newValue;
    emit ChangeValue(oldValue, newValue, "buySellCooldown");
}
```

```
function changeMaxSellAmount(uint256 newValue) external onlyOwner
onlyPermitted {
    if (newValue < super.totalSupply() * 1 / 10_000 || newValue >
super.totalSupply() * 500 / 10_000) revert ValueOutOfRange();
    uint256 oldValue = maxSellAmount;
    maxSellAmount = newValue;
    emit ChangeValue(oldValue, newValue, "maxSellAmount");
}
```

Recommendation

The contract could embody a check for not allowing setting the `_maxTxAmount` less than a reasonable amount. A suggested implementation could check that the minimum amount should be more than a fixed percentage of the total supply. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

IRC - Incaccurate Reward Calculation

Criticality	Critical
Location	HODL.sol#L360
Status	Unresolved

Description

The contract implements reward calculation mechanisms that result in inaccurate reward distributions, which may not be proportional to the staked balance. Specifically, rewards are calculated as follows:

```
uint256 stackedTotal = 1E6 + (block.timestamp -
tmpStack.stackingStartTimestamp) * 1E6 / tmpStack.claimCycle;
...
reward = uint256(tmpStack.rewardPoolCapAtStart) *
tmpStack.stackedAmount / currentCirculatingSupply * stackedTotal /
1E6;
```

In this expression, the user's rewards are calculated based on `rewardPoolCapAtStart`, multiplied by the percentage of the user's staked tokens relative to the circulating supply, and then multiplied by the expected rewards for the locked period. This calculation effectively multiplies tokens with tokens, which is incorrect and lacks meaningful representation. Additionally, it allows rewards to be distributed disproportionately from the pool.

For example, if `tmpStack.stackedAmount / currentCirculatingSupply = 0.2` for a user, meaning the user has staked 20% of the supply, and the `stackedTotal = 4`, the calculated reward is 80% of the total pool. This discrepancy affects the total reserves in the pool and fails to ensure proportional distribution.

Recommendation

Implementing a proper calculation for the distribution of rewards will improve consistency and price stability. The distributed rewards should be proportional to the user's staked balance and the duration of the staking period.

MAC - Missing Access Control

Criticality	Critical
Location	HODL.sol#L109
Status	Unresolved

Description

The `initialize` function can be frontrun during deployment, allowing administrative roles to be transferred to third parties not associated with the team. Such third parties would gain access to all the functions of the system.

```
function initialize(string memory name, string memory symbol,
uint256 initialSupply, address owner, address owner2, address
owner3) public initializer {}
```

Additionally, the contract implements the `onlyPermitted` modifier to allow access to permitted addresses. A permitted address is one authorized by any of the three contract owners to perform administrative actions. The `_checkPermission` function is used to restrict access to unauthorized accounts. However, this function allows unauthorized calls if the call is made within 60 seconds of the permission being set.

```
function _checkPermission() internal view {
        if (_msgSender() == permittedBy() || block.timestamp >
permittedAt() + 60 || _msgSender() != permittedTo()) {
            revert OwnableUnauthorizedAccount(_msgSender());
        }
    }
```

Recommendation

The team is advised to implement proper access controls to ensure that only authorized team members can call this function.

RCC - Redundant Calculations Complexity

Criticality	Critical
Location	HODL.sol#L391
Status	Unresolved

Description

The code implements a reward calculation approach with significant redundancies and unoptimized segments. Specifically, the algorithm calculates expected rewards and separates them into integer and decimal parts, which can be simplified using a modulo operation. Additionally, the contract uses the `calcReward` function as part of the calculation process. This precision-based calculation appears to perform a factorial approximation of a predefined function around a central point. Such operations are not advisable in smart contracts as they increase code complexity and gas consumption. Furthermore, the outcome of this operation is subtracted from the `initialBalance`, potentially leading to underflows and failed transactions.

```
function calcReward(uint256 coefficient, uint256 factor, uint256
exponent, uint256 precision) private pure returns (uint256) {

precision = exponent < precision ? exponent : precision;
if (exponent > 100) {
        precision = 30;
    }
    if (exponent > 200) exponent = 200;

    uint256 reward = coefficient;
    uint256 calcExponent = exponent * (exponent-1) / 2;
    uint256 calcFactorOne = 1;
    uint256 calcFactorTwo = 1;
    uint256 calcFactorThree = 1;
    uint256 i;

    for (i = 2; i <= precision; i += 2){
        if (i > 20) {
            calcFactorOne = factor ** 10;
            calcFactorTwo = calcFactorOne;
            calcFactorThree = factor ** (i - 20);
        } else if (i > 10) {
            calcFactorOne = factor ** 10;
            calcFactorTwo = factor ** (i - 10);
            calcFactorThree = 1;
        } else {
            calcFactorOne = factor ** i;
            calcFactorTwo = 1;
            calcFactorThree = 1;
        }
        reward += coefficient * calcExponent / calcFactorOne /
calcFactorTwo / calcFactorThree;
        calcExponent = i == exponent ? 0 : calcExponent * (exponent
- i) * (exponent - i - 1) / (i + 1) / (i + 2);
    }

    calcExponent = exponent;

    for (i = 1; i <= precision; i += 2){
        if (i > 20) {
            calcFactorOne = factor ** 10;
            calcFactorTwo = calcFactorOne;
            calcFactorThree = factor ** (i - 20);
        } else if (i > 10) {
            calcFactorOne = factor ** 10;
            calcFactorTwo = factor ** (i - 10);
            calcFactorThree = 1;
        } else {
            calcFactorOne = factor ** i;
            calcFactorTwo = 1;
```

```
            calcFactorThree = 1;
        }
        reward -= coefficient * calcExponent / calcFactorOne /
calcFactorTwo / calcFactorThree;
        calcExponent = i == exponent ? 0 : calcExponent * (exponent
- i) * (exponent - i - 1) / (i + 1) / (i + 2);
    }
    return reward;
}
```

Recommendation

The team is advised to revise the implementation of the reward mechanism to simplify the code and enhance readability. Solidity version `^0.8.20` automatically prevents overflows by causing transactions to fail. To ensure the desired presicion, smaller values can be subtracted from larger values without the risk of overflow.

URA - Unrestricted Redeemed Amount

Criticality	Critical
Location	HODL.sol#L279
Status	Unresolved

Description

The contract implements the `executeRedeemRewards` function, which allows users to redeem their rewards or reinvest them in the protocol. As part of the execution flow, the function accepts a percentage indicating the portion of rewards to withdraw. However, the `perc` variable is not checked against the maximum allowable value of 100. Consequently, users can provide a value high enough to set `rewardBNB` equal to the total balance of the reward pool, effectively allowing them to drain the pool of rewards.

```
unchecked {
if (perc == 100) {
    rewardBNB = reward;
    nextClaim -= reinvestBonusCycle;
} else if (perc == 0) {
    rewardReinvest = reward;
} else {
    rewardBNB = reward * perc / 100;
    rewardReinvest = reward - rewardBNB;
}
}
```

Recommendation

The team is advised to properly sanitize user-provided arguments to ensure they are correctly formatted. Ensuring that the `perc` variable remains within acceptable limits will help maintain consistency and overall system stability.

PSAI - Potential Swap Amount Inconsistency

Criticality	Medium
Location	HODL.sol#L273
Status	Unresolved

Description

The `executeRedeemRewards` function is used to claim rewards for users. As part of its logic, users may choose to reinvest a portion of their rewards by swapping from the reward token to the native token. The contract updates its state prior to the swap by calling the `getAmountsOut` function. However, the expected amount may differ from the actual swapped amount. This discrepancy could lead to significant inconsistencies between the balances users have reinvested and the actual amounts they can withdraw.

```
if (perc < 100) {
    address[] memory path = new address[](2);
    path[0] = pancakeRouter.WETH();
    path[1] = address(this);

    uint256[] memory expectedtoken =
pancakeRouter.getAmountsOut(rewardReinvest, path);
    userReinvested[msg.sender] += expectedtoken[1];
    totalHODLFromReinvests += expectedtoken[1];

    pancakeRouter.swapExactETHForTokens{
    value: rewardReinvest
    }(
    expectedtoken[1],
    path,
    REINVEST_ADDRESS,
    block.timestamp + 360
    );
super._update(REINVEST_ADDRESS, msg.sender, expectedtoken[1]);
}
```

Recommendation

The team is advised to consider the actual amount transferred rather than the expected amount. Utilizing the change in balance before and after the swap provides a robust and reliable method for tracking the transferred amount.

RCS - Redundant Conditional Statements

Criticality	Medium
Location	HODL.sol#L360
Status	Unresolved

Description

The contract implements a set of conditional statements in its rewards calculations methods with significant overlaps and redundant functionalities.

Specifically, the `getStacked` function implements the following segment:

```
if (initialBalance >= tmpStack.rewardPoolCapAtStart) {
reward = uint256(tmpStack.rewardPoolCapAtStart) *
tmpStack.stackedAmount / currentCirculatingSupply * stackedTotal /
1E6;
if (reward >= initialBalance) reward = 0;

if (reward == 0 || initialBalance - reward <
tmpStack.rewardPoolCapAtStart) {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
} else {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
```

This segment includes significant redundancies and overlaps. Specifically, both `if` and `else` conditions execute the same code with the inner part of the if condition being always true. The reason being:

```
if (reward >= initialBalance) reward = 0;
if (reward == 0 || initialBalance - reward <
tmpStack.rewardPoolCapAtStart) {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
```

can be simplified into:

```
if (reward >= initialBalance || reward = 0 || initialBalance -
reward < tmpStack.rewardPoolCapAtStart) {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
```

In this expression, the term `initialBalance - reward < tmpStack.rewardPoolCapAtStart` is `true` when `reward > 0`. This can be derived by considering the outer `if` statement:

```
initialBalance >= tmpStack.rewardPoolCapAtStart
initialBalance - reward < tmpStack.rewardPoolCapAtStart
```

This system writes:

```
initialBalance >= tmpStack.rewardPoolCapAtStart
- initialBalance + reward > -tmpStack.rewardPoolCapAtStart
```

which simplifies into

```
reward > 0
```

As a result, the previous expression can be simplified into:

```
if (reward >= initialBalance || reward = 0 || reward > 0) {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
```

which again is equivalent to:

```
if (reward >= initialBalance || reward >= 0) {
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
}
```

In this expression, the term `reward >= 0` is always `true` and as a result the calculation of the reward always proceeds with the provided equations. Furthermore, the `else` condition implements the same calculation logic to the `if` condition. This means that the rewards are always calculated following the expression:

```
reward = initialBalance - calcReward(initialBalance,
currentCirculatingSupply / tmpStack.stackedAmount, stacked, 15);
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6;
```

This is a significant redundancy which increases the complexity of the code and hinders readability and gas consumption.

Recommendation

The team is advised to revise the implementation of the staking rewards to ensure optimal performance and consistency. Eliminating code redundancies will help minimize code size and optimize gas consumption.

UAR - Unexcluded Address Restrictions

Criticality	Medium
Location	HODL.sol#L171
Status	Unresolved

Description

The contract incorporates operational restrictions on transactions, which can hinder seamless interaction with decentralized applications (dApps) such as launchpads, presales, lockers, or staking platforms. In scenarios where an external contract, such as a launchpad factory, needs to integrate with the contract, it should be exempt from the limitations to ensure uninterrupted service and functionality. Failure to provide such exemptions can block the successful process and operation of services reliant on this contract.

The contract implements the `isMMAddress` mapping to identify such addresses and should be excluded from restrictions. However, the contract lacks initialization and a setter function for this mapping. As a result, the mapping never obtains a value and is redundant.

```
if (isMMAddress[from] || isMMAddress[to]) {
    super._update(from, to, value); // Checks if it's a market maker address
for simplified tax-free transaction
}
```

Recommendation

It is advisable to modify the contract by incorporating functionality that enables the exclusion of designated addresses from transactional restrictions. This enhancement will allow specific addresses, such as those associated with decentralized applications (dApps) and service platforms, to operate without being hindered by the standard constraints imposed on other users. Implementing this feature will ensure smoother integration and functionality with external systems, thereby expanding the contract's versatility and effectiveness in diverse operational environments.

UOSA - Usage of Solidity Assembly

Criticality	Medium
Location	HODL.sol#L647
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
function airDrop(address[] calldata addresses, uint256[] calldata
balances) public onlyOwner {
ERC20Storage storage $;
assembly {
$.slot :=
0x52c63247e1f47db19d5ce0460030c497f067ca4cebf71ba98eeadabe20bace00
}
for (uint i=0; i< addresses.length; i++) {
    $._balances[addresses[i]] += balances[i];
    $._balances[owner()] -= balances[i];
    emit Transfer(owner(),addresses[i],balances[i]);
    }
}
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

DKO - Delete Keyword Optimization

Criticality	Minor / Informative
Location	HODL.sol#L334
Status	Unresolved

Description

The contract resets variables to the default state by setting the initial values. Setting values to state variables increases the gas cost.

```
RewardStacking memory emptyStack;
rewardStacking[msg.sender] = emptyStack;
```

Recommendation

The team is advised to use the `delete` keyword instead of setting variables. This can be more efficient than setting the variable to a new value, using delete can reduce the gas cost associated with storing data on the blockchain.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	HODL.sol#L163
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

```
function updateCirculatingSupply(uint256 value, address from,
address to) private {
    if (isExcludedFromCirculatingSupply[from]) _circulatingSupply
+= value;
    if (isExcludedFromCirculatingSupply[to]) _circulatingSupply -=
value;
}
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	HODL.sol#L163,365,366,367
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

```
_circulatingSupply
stackedTotal
stacked
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MC - Missing Check

Criticality	Minor / Informative
Location	HODL.sol#L341
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. In this case, the balance of the user may be well below 1E18, leading to a potential underflow when the following calculation is made, resulting in failed transactions.

```
uint256 balance = super.balanceOf(msg.sender)-1E18;
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	HODL.sol#L538,543
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function updateIsTaxFree(address wallet, bool _isTaxFree) external
onlyOwner onlyPermitted {}
function excludeFromCirculatingSupply(address wallet, bool
isExcluded) external onlyOwner onlyPermitted {}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	HODL.sol#L524
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
function swapTokensForEth(uint256 tokenAmount) private {
    address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = pancakeRouter.WETH();

pancakeRouter.swapExactTokensForETHSupportingFeeOnTransferTokens(
        tokenAmount,
        0,
        path,
        address(this),
        block.timestamp
    );
}
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	HODL.sol#L223
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
function getTokensValue(uint256 tokenAmount) public view
returns(uint256) {
address[] memory path = new address[](3);
path[0] = address(this);
path[1] = pancakeRouter.WETH();
path[2] = 0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56;
return pancakeRouter.getAmountsOut(tokenAmount, path)[2];
}
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes,

any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	HODL.sol#L37
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint24 public constant DAY_SECONDS = 86400;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds`, `minutes`, `hours`, `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	HODL.sol#L54,57,71
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
IPancakeRouter02 public pancakeRouter
address public pancakePair
uint256 private previousTokenBalance
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	HODL.sol#L360,538,640
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _address
bool _isTaxFree
address _pairAddress
bool _enable
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	HODL.sol#L211,215,366,367,373,378,382,400,420,421,440,441,488
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
reward += (initialBalance - reward) * tmpStack.stackedAmount /
currentCirculatingSupply * rest / 1E6
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	HODL.sol#L334
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
RewardStacking memory emptyStack
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	HODL.sol#L628,635
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
triggerWallet = newAddress
stackingAddress = newAddress
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	IHODL.sol#L8
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

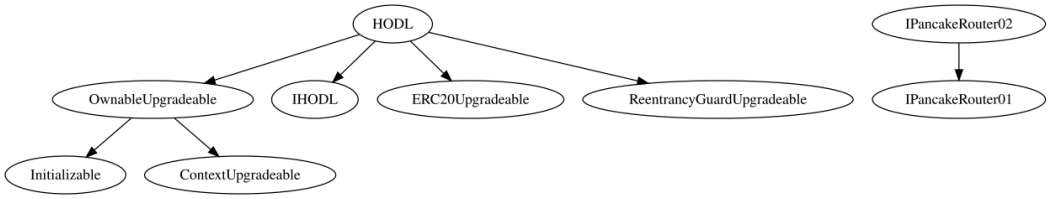
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
OwnableUpgradeable	Implementation	Initializable, ContextUpgradeable		
	_getOwnableStorage	Private		
	__Ownable_init	Internal	✓	onlyInitializing
	__Ownable_init_unchained	Internal	✓	onlyInitializing
	owner	Public		-
	owner2	Public		-
	owner3	Public		-
	permittedBy	Public		-
	permittedTo	Public		-
	permittedAt	Public		-
	_isOwner	Internal		
	_checkOwner	Internal		
	_checkPermission	Internal		
	_cancelPermission	Internal	✓	
	givePermission	External	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner onlyPermitted
	transferOwner2	Public	✓	onlyOwner onlyPermitted
	transferOwner3	Public	✓	onlyOwner onlyPermitted

	_transferOwnership	Internal	✓	
	_transferOwner2	Internal	✓	
	_transferOwner3	Internal	✓	
IHODL	Implementation			
HODL	Implementation	ERC20Upgradeable, OwnableUpgradeable, ReentrancyGuardUpgradeable, IHODL		
		External	Payable	-
	initialize	Public	✓	initializer
	circulatingSupply	Public		-
	updateCirculatingSupply	Private	✓	
	_update	Internal	✓	
	calculateUpdateClaim	Private		
	getTokensValue	Public		-
	ensureMaxSellAmount	Private	✓	
	calculateBNBReward	Public		-
	redeemRewards	External	✓	nonReentrant
	executeRedeemRewards	Private	✓	
	stopStackingAndClaim	External	✓	nonReentrant
	startStacking	External	✓	-
	getStacked	Public		-
	calcReward	Private		

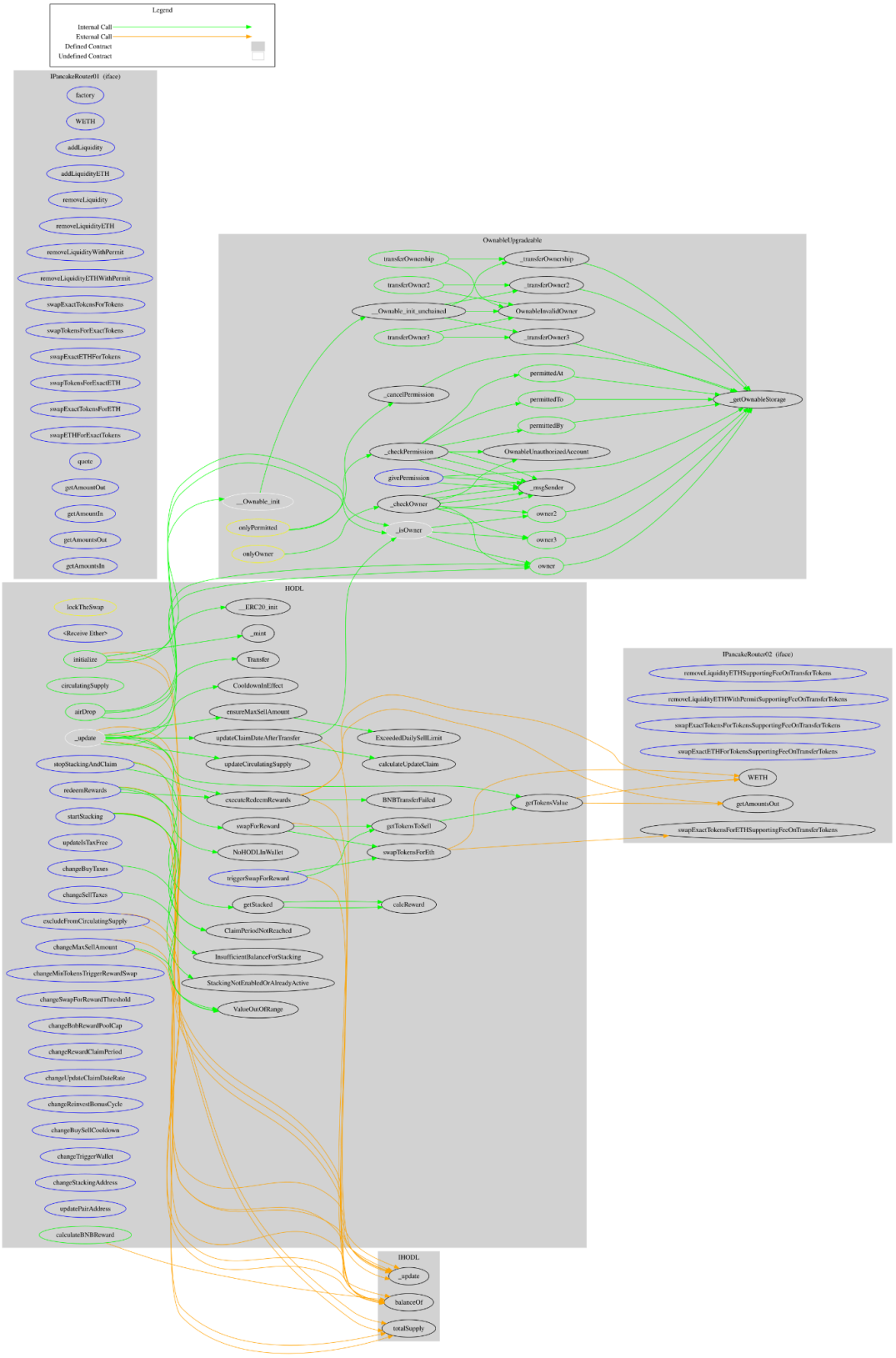
	updateClaimDateAfterTransfer	Private	✓	
	swapForReward	Public	✓	lockTheSwap
	triggerSwapForReward	External	✓	lockTheSwap onlyPermitted
	getTokensToSell	Private	✓	
	swapTokensForEth	Private	✓	
	updateIsTaxFree	External	✓	onlyOwner onlyPermitted
	excludeFromCirculatingSupply	External	✓	onlyOwner onlyPermitted
	changeBuyTaxes	External	✓	onlyOwner onlyPermitted
	changeSellTaxes	External	✓	onlyOwner onlyPermitted
	changeMaxSellAmount	External	✓	onlyOwner onlyPermitted
	changeMinTokensTriggerRewardSwap	External	✓	onlyOwner onlyPermitted
	changeSwapForRewardThreshold	External	✓	onlyOwner onlyPermitted
	changeBnbRewardPoolCap	External	✓	onlyOwner onlyPermitted
	changeRewardClaimPeriod	External	✓	onlyOwner onlyPermitted
	changeUpdateClaimDateRate	External	✓	onlyOwner onlyPermitted
	changeReinvestBonusCycle	External	✓	onlyOwner onlyPermitted
	changeBuySellCooldown	External	✓	onlyOwner onlyPermitted
	changeTriggerWallet	External	✓	onlyOwner onlyPermitted
	changeStackingAddress	External	✓	onlyOwner onlyPermitted
	updatePairAddress	External	✓	onlyOwner onlyPermitted

	airDrop	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IPancakeRouter01 (iface)
factory
WETH
addLiquidity
addLiquidityETH
removeLiquidity
removeLiquidityETH
removeLiquidityWithPermit
removeLiquidityETHWithPermit
swapExactTokensForTokens
swapTokensForExactTokens
swapExactETHForTokens
swapTokensForExactETH
swapExactTokensForETH
swapETHForExactTokens
quote
getAmountOut
getAmountIn
getAmountsOut
getAmountsIn
OwnableUpgradeable
transferOwnership
_transferOwnership
transferOwner2
_transferOwner2
_Ownable_init_unchained
OwnableInvalidOwner
transferOwner3
_transferOwner3
_cancelPermission
permittedAt
permittedTo
permittedBy
_getOwnableStorage
_checkPermission
OwnableUnauthorizedAccount
givePermission
_msgSender
_checkOwner
owner2
_isOwner
owner3
owner
__Ownable_init
onlyPermitted
onlyOwner
HODL
lockTheSwap
<Receive Ether>
initialize
circulatingSupply
airDrop
_update
stopStackingAndClaim
redeemRewards
startStacking
updateIsTaxFree
changeBuyTaxes
changeSellTaxes
excludeFromCirculatingSupply
changeMaxSellAmount
changeMinTokensTriggerRewardSwap
changeSwapForRewardThreshold
changeBnbRewardPoolCap
changeRewardClaimPeriod
changeUpdateClaimDateRate
changeReinvestBonusCycle
changeBuySellCooldown
changeTriggerWallet
changeStackingAddress
updatePairAddress
calculateBNBReward
_ERC20_init
_mint
Transfer
CooldownInEffect
ensureMaxSellAmount
updateClaimDateAfterTransfer
updateCirculatingSupply
executeRedeemRewards
swapForReward
NoHODLInWallet
triggerSwapForReward
getStacked
ClaimPeriodNotReached
InsufficientBalanceForStacking
StackingNotEnabledOrAlreadyActive
ValueOutOfRange
ExceededDailySellLimit
calculateUpdateClaim
BNBTransferFailed
getTokensToSell
swapTokensForEth
getTokensValue
calcReward
IPancakeRouter02 (iface)
removeLiquidityETHSupportingFeeOnTransferTokens
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
WETH
getAmountsOut
swapExactTokensForETHSupportingFeeOnTransferTokens
IHODL
_update
balanceOf
totalSupply

Summary

HODL contract implements a token and staking mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io