Cyberscope

Audit Report

Hodl

December 2024

SHA256

0becdc0f9d0620b49697a6739b699390ec3614e4ec983de0fc208c14c8e2dd81

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
●	IBS	Inconsistent Balance Swap	Unresolved
●	RUT	Restricted User Transactions	Unresolve
●	UTS	Uninitialized Token Supply	Unresolved
●	IMRA	Inconsistent Maximum Reward Amount	Unresolved
●	DRA	Dynamic Reward Allocation	Unresolved
●	MMN	Misleading Method Naming	Unresolved
●	MC	Missing Check	Unresolved
●	MMRR	Missing Maximum Reward Restriction	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PVC	Price Volatility Concern	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 https://github.com/cyberscope-io/audits/blob/main/hodl/v1/audit.pdf
Corrected Phase 2	05 Dec 2024
Test Deploy	https://sepolia.etherscan.io/address/0x38037D7834B47E44a3a6A67586B7Bdb0c2DA6199

Source Files

Filename	SHA256
OwnableUpgradeable.sol	ebf38dc17b401ac3a98de2db8c31e184b52dac2c7e8ac5e53884298a8f815a0c
IHODL.sol	67634ec775564454928c13c15532201321f9c812ca130738c4cc02edd731ad7d
HODL.sol	0becdc0f9d0620b49697a6739b699390ec3614e4ec983de0fc208c14c8e2dd81

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.

Overview

The Hodl contract implements a token mechanism with staking and reward distribution functionalities. Users can claim rewards collected by the contract in the form of fees. Stakers can also claim from the reward pool by compounding their share. Additionally, users may choose to reinvest their rewards by swapping them for Hodl tokens. Main functionalities include:

startStacking Function:

This function enables staking for a user and stakes all of their token balance except for 1 token. The contract checks that the user does not have an active staking session and that the balance exceeds a minimum threshold. If that is true, the user's balance is transferred to the contract and the staking information is stored in a structure including the staked amount, the timestamp at the start of the staking and the claim period.

redeemRewards Function:

The contract allows a holder of the token to claim a percentage of the reward pool based on their token balance . Specifically, if the current balance of the contract exceeds a threshold, the user's rewards are calculated as a percentage of that threshold, proportional to their balance as a percentage of the circulating supply. Otherwise, if the contract's balance is less than the threshold, the former is used. It is important to notice that the user's balance is used for the calculation of rewards, rather than the staked balance. The redeemRewards function can be therefore called by external addresses that do not necessarily stake the token.

stopStackingAndClaim Function:

This function allows users to terminate their staking and claim their rewards. Rewards are calculated through a call to the getStacked function. Here the contract considers several cases.

Initially, if the contract holds more tokens than the cap set for the claimable rewards, the latter is used in the calculations.

In this case, rewards are estimated as a portion of the cap proportionally to the staked balance of the user as a percentage of the circulating supply, multiplied by the integer of periods the user has been staking.

$$rewards = rewardCap * \frac{userStaked}{circulatingSupply} * stakedPeriods$$

In this calculation, it is possible that the rewards exceed the balance of the contract.

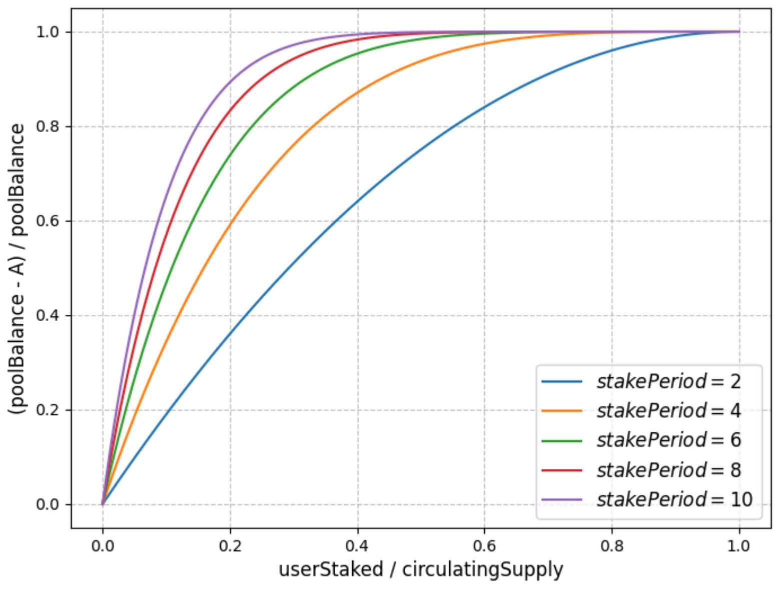
In this case, the rewards are re-evaluated by an interesting but rather complicated distribution mechanism which is inspired by the compound interest methodology. Specifically, the contract utilizes a series approximation to calculate the equation:

$$A = poolBalance * (1 - userStaked/circulatingSupply)^{stakedPeriods}$$

This calculation returns the portion of the pool balance that is inaccessible to the user. To estimate the portion that the user may claim, A is subtracted from the pool balance:

$$reward = poolBalance - A$$

The latter estimates the claimable rewards as a function of the staked amount, the staking period and the pool size. In the following graph, the available rewards as a portion of the pool size are plotted for different magnitudes of staked amounts and staked durations:

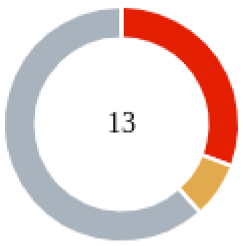


In this figure, a larger portion of the pool becomes available, on the y-axis, as the staked amount or the staking period increases. Specifically, for a constant staked amount, more rewards are unlocked as time passes, while for the same staking period, larger staked amounts yield larger rewards.

The graph reveals that multiple configurations allow the estimated rewards to converge to the total pool becoming available. Users therefore, need to elect the proper strategy to maximize their profits.

Furthermore, these rewards are adjusted by a percentage of the pool amount that remains inaccessible proportionally to the user's staked balance and the elapsed time from the current period. At the end of these calculations a hard threshold is applied on the estimated rewards ensuring that they cannot exceed a predefined limit, set at the start of the staking.

Findings Breakdown



● Critical	4
● Medium	1
● Minor / Informative	8

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	4	0	0	0
● Medium	1	0	0	0
● Minor / Informative	8	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	HODL.sol#L196
Status	Unresolved

Description

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 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.

IBS - Inconsistent Balance Swap

Criticality	Critical
Location	HODL.sol#L346
Status	Unresolved

Description

The contract swaps collected fees for rewards, which can be claimed by token holders. The swap is executed during sales when the sold value exceeds a `swapForRewardThreshold` , triggering the `swapForReward` function. This function calculates the contract's current token balance and its reserves in native tokens. If the token balance exceeds a threshold and the contract holds fewer native tokens than the cap for rewards, a portion of the contract's token reserves is exchanged for the native token. However, the contract fails to ensure the staked amounts are preserved among the token reserves, potentially swapping staked tokens for rewards. As a result, users may be unable to withdraw their deposits.

```
function _update(address from, address to, uint256 value) internal
virtual override {
...
if (rewardSwapEnabled && !_inRewardSwap && getTokensValue(value) >
swapForRewardThreshold) {
swapForReward(from, to);
}
...
}
```

```
function swapForReward(address from, address to) private
lockTheSwap {
uint256 contractTokenBalance = super.balanceOf(address(this));
uint256 currentPoolBalance = address(this).balance;
// Trigger reward swap if pool balance meets threshold and is below
cap
if (contractTokenBalance >= minTokensTriggerRewardSwap &&
currentPoolBalance <= bnbRewardPoolCap &&
from != PANCAKE_PAIR &&
!(from == address(this) && to == address(PANCAKE_PAIR))
) {
uint256 tokensToSell = getTokensToSell(currentPoolBalance,
contractTokenBalance);
if (tokensToSell > 0) swapTokensForEth(tokensToSell);
}
}
```

Recommendation

The contract should distinguish between the staked amounts and the fees collected to be distributed for rewards. Failure to do so could result in loss of funds for the impacted users.

RUT - Restricted User Transactions

Criticality	Critical
Location	HODL.sol#L331
Status	Unresolved

Description

In smart contracts, the `_update` method is often used to transfer funds between accounts. In the current implementation, the contract includes a conditional check to ensure that either the sender or the recipient of the transaction is on the list of owners. This constraint restricts users from transferring the token, making normal transactions impossible. Additionally, there is no way to bypass or deactivate this restriction for users who are not enrolled in the owner list.

```
require(_isOwner(from) || _isOwner(to), "Contract paused while initial
setup!");
```

Recommendation

Removing this restriction or implementing functionalities that allow it to be deactivated would resolve this issue and enable the normal operation of the token.

UTS - Uninitialized Token Supply

Criticality	Critical
Location	HODL.sol
Status	Unresolved

Description

The contract implements a token mechanism that supports functionalities like staking and claiming rewards. However, the contract fails to initialize the token supply. As a result, the total supply of the token is set to zero, rendering all contract functionalities ineffective.

Recommendation

The total supply of the token can be initialized during deployment by calling the internal `_mint` function within a constructor. For more information, the team is referred to the OpenZeppelin documentation.
https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20-_mint-address-uint256-

DRA - Dynamic Reward Allocation

Criticality	Minor / Informative
Location	HODL.sol#L297,395
Status	Unresolved

Description

The contract distributes rewards from a pool of accumulated fees. These rewards are finite and can be claimed at any time by any eligible address. At the time of the claim, rewards are calculated based on the available pool balance or a cap value. In the first case, race conditions and optimization points are induced where users are incentivized to claim rewards before the pool is depleted. This may disincentivize users from compounding their rewards through longer stakes. While it could also result in unexpected redeemable rewards as the pool balance changes over time.

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

Recommendation

Implementing an economic design that ensures proportional distribution according to the staked balance and period, while guaranteeing that the expected amount accumulates independently of changes in the reward balance, will enhance consistency and user trust in the system. The team is advised to implement a mechanism that updates the user's rewards at the time the balance is updated.

IMRA - Inconsistent Maximum Reward Amount

Criticality	Medium
Location	HODL.sol#L147,525
Status	Unresolved

Description

The contract implements the `redeemRewards` function, which allows the caller to redeem rewards from the contract's reward reserves. In this calculation, the user's amount is estimated based on the caller's current balance, not the actual staked amount. Rewards are calculated proportionally to the balance as a function of the circulating supply. This approach disincentivizes users from staking in the contract, as rewards can be claimed simply by holding tokens. Additionally, it allows users to transfer funds between their own accounts to exploit favorable staking claim dates without triggering the calculation of a `newCycleBlock` in the new wallet.

```
function redeemRewards(uint8 perc) external nonReentrant {
if (perc > 100) revert ValueOutOfRange();
uint256 userBalance = super.balanceOf(msg.sender);
if (nextClaimDate[msg.sender] > block.timestamp) revert
ClaimPeriodNotReached();
if (userBalance == 0) revert NoHODLInWallet();
uint256 currentBNBPool = address(this).balance;
uint256 reward = currentBNBPool > bnbRewardPoolCap ?
bnbRewardPoolCap * userBalance / circulatingSupply : currentBNBPool
* userBalance / circulatingSupply;
executeRedeemRewards(perc, reward);
}
```

Recommendation

It is advisable to ensure that rewards are claimable only by staked accounts to maintain consistency and fairness in the reward distribution mechanism. Additionally, ensuring the proper application of claiming restrictions across all addresses will prevent potential manipulation of the reward pool.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	HODL.sol#L370
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.

MC - Missing Check

Criticality	Minor / Informative
Location	HODL.sol#L128
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)1 ether;;
```

Recommendation

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

MMRR - Missing Maximum Reward Restriction

Criticality	Minor / Informative
Location	HODL.sol#L147
Status	Unresolved

Description

The contract implements two methods for withdrawing rewards from the pool's reserves: by calling the `redeemRewards` function or by using the `stopStackingAndClaim` function to withdraw staked amounts and claim available rewards. In the latter case, a maximum reward amount is always applied, equal to a `rewardLimit` initialized at the start of staking. However, in the first case, this limit is not imposed, allowing users to claim rewards exceeding the limit.

```
function redeemRewards(uint8 perc) external nonReentrant {
if (perc > 100) revert ValueOutOfRange();
uint256 userBalance = super.balanceOf(msg.sender);
if (nextClaimDate[msg.sender] > block.timestamp) revert
ClaimPeriodNotReached();
if (userBalance == 0) revert NoHODLInWallet();
uint256 currentBNBPool = address(this).balance;
uint256 reward = currentBNBPool > bnbRewardPoolCap ?
bnbRewardPoolCap * userBalance / circulatingSupply : currentBNBPool
* userBalance / circulatingSupply;
executeRedeemRewards(perc, reward);
}
```

Recommendation

The team is advised to ensure consistency between the different claiming mechanisms. Calculating rewards using the same formula and restrictions will ensure the smooth operation of the rewards pool.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	HODL.sol#L511
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#L288
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.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	HODL.sol#L211
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapForRewardThreshold` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

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

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

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

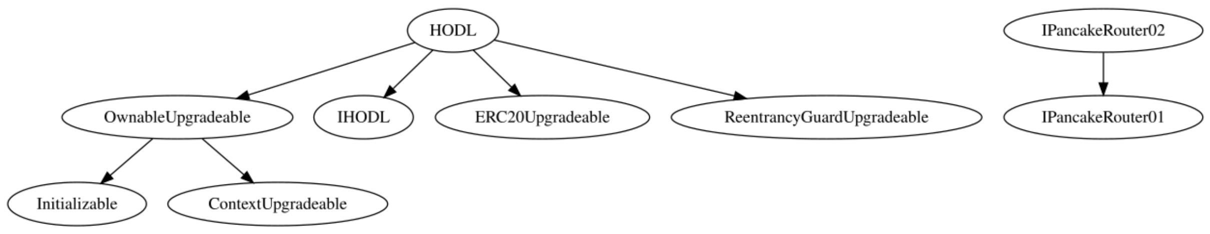
	_transferOwnership	Internal	✓	
	_transferOwner2	Internal	✓	
	_transferOwner3	Internal	✓	
IPancakeRouter02	Interface	IPancakeRouter01		
	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
IPancakeRouter01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-

	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IHODL	Implementation			
HODL	Implementation	ERC20Upgradeable, OwnableUpgradeable, ReentrancyGuardUpgradeable, IHODL		
		External	Payable	-
	upgrade	External	✓	onlyOwner reinitializer
	stopStackingAndClaim	External	✓	nonReentrant
	startStacking	External	✓	-
	redeemRewards	External	✓	nonReentrant
	updateIsTaxFree	External	✓	onlyOwner onlyPermitted
	excludeFromCirculatingSupply	External	✓	onlyOwner onlyPermitted
	changeBuyTaxes	External	✓	onlyOwner onlyPermitted
	changeSellTaxes	External	✓	onlyOwner onlyPermitted

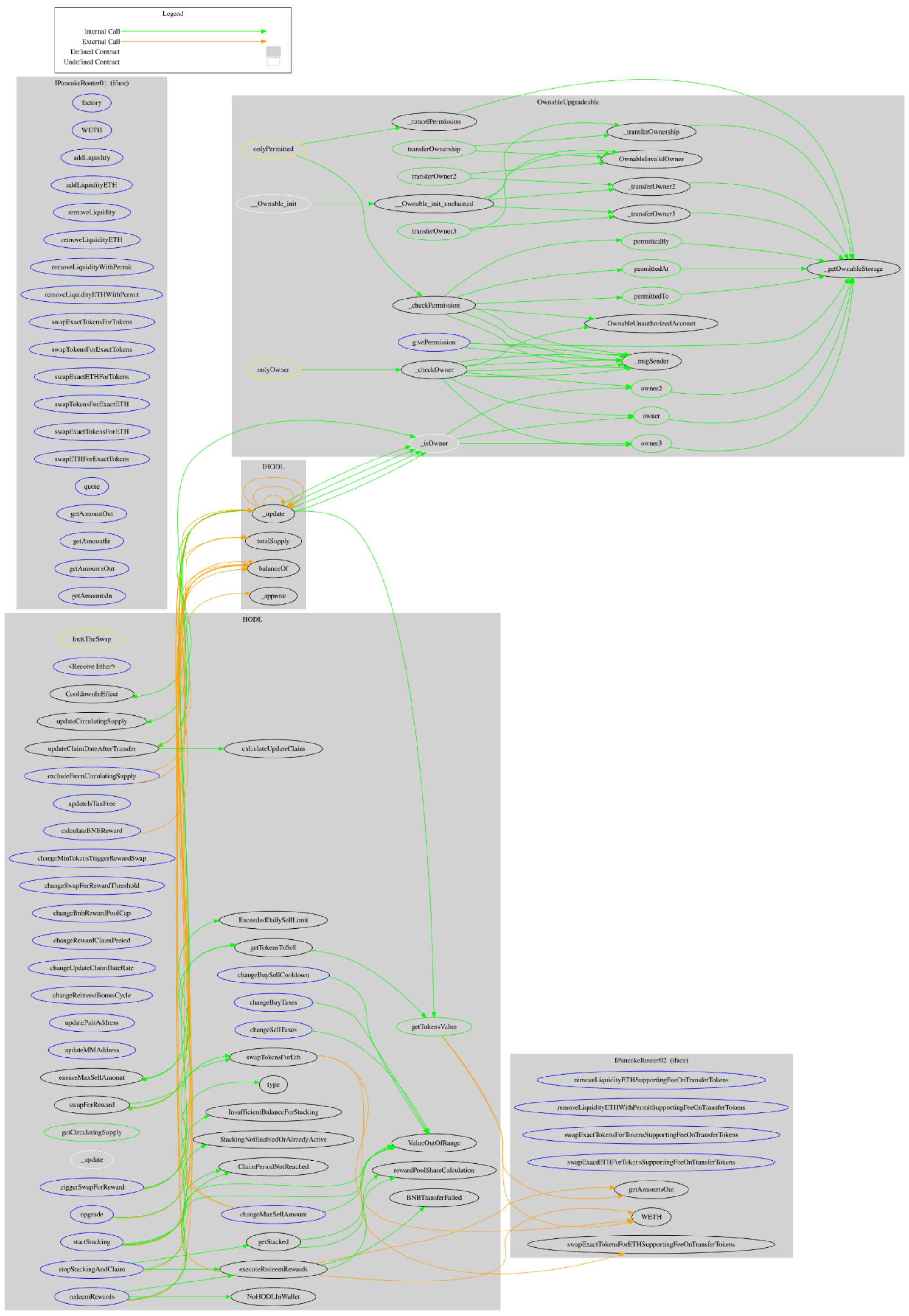
	changeMaxSellAmount	External	✓	onlyOwner onlyPermitted
	changeMinTokensTriggerRewardSwap	External	✓	onlyOwner onlyPermitted
	changeSwapForRewardThreshold	External	✓	onlyOwner onlyPermitted
	changeBnbRewardPoolCap	External	✓	onlyOwner onlyPermitted
	changeRewardClaimPeriod	External	✓	onlyOwner onlyPermitted
	changeUpdateClaimDateRate	External	✓	onlyOwner onlyPermitted
	changeReinvestBonusCycle	External	✓	onlyOwner onlyPermitted
	changeBuySellCooldown	External	✓	onlyOwner onlyPermitted
	updatePairAddress	External	✓	onlyOwner onlyPermitted
	updateMMAddress	External	✓	onlyOwner onlyPermitted
	triggerSwapForReward	External	✓	lockTheSwap onlyPermitted
	calculateBNBReward	External		-
	getCirculatingSupply	Public		-
	getTokensValue	Public		-
	getStacked	Public		-
	_update	Internal	✓	
	updateCirculatingSupply	Private	✓	
	ensureMaxSellAmount	Private	✓	
	executeRedeemRewards	Private	✓	
	updateClaimDateAfterTransfer	Private	✓	
	swapForReward	Private	✓	lockTheSwap

	getTokensToSell	Private	✓	
	swapTokensForEth	Private	✓	
	calculateUpdateClaim	Private		
	rewardPoolShareCalculation	Private		

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
onlyPermitted
_cancelPermission
transferOwnership
transferOwner2
__Ownable_init
__Ownable_init_unchained
transferOwner3
_transferOwnership
OwnableInvalidOwner
_transferOwner2
_transferOwner3
permittedBy
permittedAt
permittedTo
_getOwnableStorage
_checkPermission
OwnableUnauthorizedAccount
givePermission
onlyOwner
_checkOwner
_msgSender
owner2
owner
owner3
_isOwner
IHODL
_update
totalSupply
balanceOf
_approve
HODL
lockTheSwap
<Receive Ether>
CooldownInEffect
updateCirculatingSupply
updateClaimDateAfterTransfer
calculateUpdateClaim
excludeFromCirculatingSupply
updateIsTaxFree
calculateBNBReward
changeMinTokensTriggerRewardSwap
changeSwapForRewardThreshold
changeBnbRewardPoolCap
changeRewardClaimPeriod
changeUpdateClaimDateRate
changeReinvestBonusCycle
updatePairAddress
updateMMAddress
ensureMaxSellAmount
swapForReward
getCirculatingSupply
_update
triggerSwapForReward
upgrade
startStacking
stopStackingAndClaim
redeemRewards
ExceededDailySellLimit
getTokensToSell
changeBuySellCooldown
changeBuyTaxes
changeSellTaxes
swapTokensForEth
type
InsufficientBalanceForStacking
StackingNotEnabledOrAlreadyActive
ClaimPeriodNotReached
changeMaxSellAmount
getStacked
executeRedeemRewards
NoHODLInWallet
getTokensValue
ValueOutOfRange
rewardPoolShareCalculation
BNBTransferFailed
IPancakeRouter02 (iface)
removeLiquidityETHSupportingFeeOnTransferTokens
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
getAmountsOut
WETH
swapExactTokensForETHSupportingFeeOnTransferTokens

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

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