Cyberscope

Audit Report

# CHAMP

April 2025

SHA256 :
fdda54ed3cb385155a03bd74b7393d973be917ebc6b70117aeedf7e6166dfd2c

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 |
|---|---|---|---|
| ● | ISA | Inconsistent Swap Amount | Unresolved |
| ● | FLV | Flash Loan Vulnerability | Unresolved |
| ● | PLPI | Potential Liquidity Provision Inadequacy | Unresolved |
| ● | TSI | Tokens Sufficiency Insurance | Unresolved |
| ● | ILT | Inconsistent Liquidity Tax | Unresolved |
| ● | MCM | Misleading Comment Messages | Unresolved |
| ● | MVN | Misleading Variables Naming | Unresolved |
| ● | MC | Missing Check | Unresolved |
| ● | PMRM | Potential Mocked Router Manipulation | Unresolved |
| ● | PVC | Price Volatility Concern | Unresolved |
| ● | RRA | Redundant Repeated Approvals | Unresolved |
| ● | UA | Unsanitized Allocation | Unresolved |
| ● | ZD | Zero Division | Unresolved |
| ● | L04 | Conformance to Solidity Naming Conventions | Unresolved |

| | | | |
|---|---|---|---|
| ● | L07 | Missing Events Arithmetic | Unresolved |
| ● | L13 | Divide before Multiply Operation | Unresolved |
| ● | L16 | Validate Variable Setters | 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

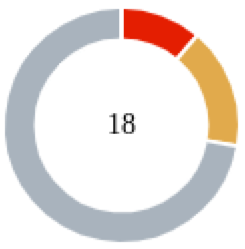
## Audit Updates

| | |
|---|---|
| **Initial Audit** | 09 Mar 2025<br>https://github.com/cyberscope-io/audits/blob/main/4-ccg/v1/audit.pdf |
| **Corrected Phase 2** | 21 Mar 2025<br>https://github.com/cyberscope-io/audits/blob/main/4-ccg/v2/audit.pdf |
| **Corrected Phase 3** | 22 Apr 2025 |
| **Test Deploys** | https://sepolia.etherscan.io/address/0x0c133bfBc275a7698dac012CE873E9C735681B9d |

## Source Files

| **Filename** | SHA256 |
|---|---|
| **CryptoChamps.sol** | fdda54ed3cb385155a03bd74b7393d973be917ebc6b70117aeedf7e6166dfd2c |

## Findings Breakdown



| | | |
|---|---|---|
| ● | Critical | 2 |
| ● | Medium | 3 |
| ● | Minor / Informative | 13 |

| Severity | | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|---|
| ● | Critical | 2 | 0 | 0 | 0 |
| ● | Medium | 3 | 0 | 0 | 0 |
| ● | Minor / Informative | 13 | 0 | 0 | 0 |

## ST - Stops Transactions

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | CryptoChamps.sol#L117 |
| **Status** | Unresolved |

### Description

The contract owner has the authority to stop all transactions for all users. The owner may take advantage of it by calling the `setTaxAllocations` function and setting a large value. Specifically, as described in finding `UA` the owner may set `liquidityAllocation` to a large value. This can potential impact all transfers by inducing an underflow in the calculation of the `reflectionTax` during the execution of the `_update` method.

```
function _update(
address from,
address to,
uint256 amount
) internal override {
uint256 contractTokenBalance = balanceOf(address(this));
bool canSwap = contractTokenBalance >= swapTokensAtAmount;

if (canSwap && !_swapping) {
_swapping = true;
uint256 liquidityTax = ((contractTokenBalance *
liquidityAllocation) / (buyTax + sellTax));
uint256 reflectionTax = contractTokenBalance - liquidityTax;
...
}
```

### Recommendation

The contract could embody a check for not allowing setting the `liquidityAllocation` to an excessive value.

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.

## ISA - Inconsistent Swap Amount

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | CryptoChamps.sol#L134 |
| **Status** | Unresolved |

### Description

As part of its transfer flow, the contract swaps tokens for the native currency and uses the proceeds to provide liquidity. Specifically, it swaps its entire balance except for a retained portion denoted as `liquidityHalf`. It then attempts to add liquidity using `liquidityHalf` and `1%` of the received native currency. However, this ratio may not correspond to the actual token-native currency ratio of the liquidity pool. Consequently, the liquidity provision may fail, causing the entire transaction to revert.

```
function _update(
address from,
address to,
uint256 amount
) internal override {
...
if (canSwap && !_swapping) {
_swapping = true;
uint256 liquidityTax = ((contractTokenBalance *
liquidityAllocation) / (buyTax + sellTax));
uint256 reflectionTax = contractTokenBalance - liquidityTax;
uint256 liquidityHalf = liquidityTax / 2;
uint256 swapHalf = liquidityTax - liquidityHalf;
uint256 tokensToSwap = swapHalf + reflectionTax;
uint256 wethOutFromSwap = _swapTokensForWETH(tokensToSwap);
uint256 percent = (wethOutFromSwap * 1) / 100;
uint256 wethUsedInLiquidity = _addToLiquidity(percent, swapHalf);
_distributeReflections(wethOutFromSwap - wethUsedInLiquidity);
_swapping = false;
}
...
}
```

## Recommendation

The team is advised to revise the implementation to ensure optimal operations. Specifically, it is recommended to swap half of the tokens allocated for liquidity into native tokens. Then, record the incoming amount of native tokens as the difference in the contract balance before and after the swap, and use that amount to provide liquidity. The amount of native tokens provided for liquidity should correspond to the actual amount of tokens received during the swap. This way, provision at the expected ratio can be ensured.

## FLV - Flash Loan Vulnerability

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | CryptoChamps.sol#L251 |
| **Status** | Unresolved |

### Description

The `calculateETHClaimable` function is susceptible to flash-loan attacks. Specifically, the function calculates the `reflectionShare` based on the user balance. A flash loan allows a user to borrow a large amount of tokens from a liquidity pool, perform operations, and return them within the same transaction. In this scenario, a user could borrow a large amount of tokens, to increase their balance. The use of `noFlashLoans` does not suffice to prevent the attack, as the caller of the method will be the same `EOA` that requested the flash loan.

```
function calculateETHWillBeClaimable(
address holder
) public view returns (uint256) {
uint256 holderBalance = balanceOf(holder);
if (holderBalance < minimumHoldingForReflection) {
return 0;
}
uint256 taxBalance = balanceOf(address(this));
uint256 withoutLiquidity = (taxBalance * 100) /
reflectionAllocation;
(uint256 willReceived, ) = _wETHAmountAndPath(withoutLiquidity);
uint256 totalSupplyExcludingBurned = totalSupply() -
balanceOf(address(0));
uint256 reflectionShare = (holderBalance *
(totalReflectionsAccumulated + willReceived)) /
totalSupplyExcludingBurned;
uint256 alreadyClaimed = totalEthReflections[holder];
return reflectionShare - alreadyClaimed;
}
```

## Recommendation

The team is advised to revise the implementation of the reward distribution mechanism to ensure secure operations.

## PLPI - Potential Liquidity Provision Inadequacy

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | CryptoChamps.sol#L154,197 |
| **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 _swapTokensForWETH(
uint256 tokenAmount
) private nonReentrant returns (uint256) {
if (tokenAmount == 0) return 0;
(uint112 r0, uint112 r1, ) = IUniswapV2Pair(liquidityPool)
.getReserves();
require(r0 > 1e18 && r1 > 1e18, "No liquidity");
...
}
```

### 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.

## TSI - Tokens Sufficiency Insurance

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | CryptoChamps.sol#L232 |
| **Status** | Unresolved |

### Description

The contract implements a reward distribution mechanism to allocate accumulated fees to users based on their balance. It tracks the accumulated funds using a `totalReflectionsAccumulated` variable. However, this variable is not consistently updated to reflect the contract's balance at all times. As a result, the contract may not maintain the necessary balance to process the withdrawal of the estimated funds.

```
function _distributeReflections(uint256 amount) private {
totalReflectionsAccumulated += amount;
emit ReflectionsDistributed(amount);
}
```

Specifically consider the following case:

```
totalReflectionsAccumulated = 10
UserA_share = 20%
UserA_share = 80%

UserA_claim = 2
UserB_claim = 8
```

Then assume rewards are claimed a second time with:

```
totalReflectionsAccumulated = 20
UserA_share = 70%
UserA_share = 30%

UserA_claim = 20*0.7-2 = 12
UserB_claim = 20*0.3-8 = -2
```

In this case the calculated amount results in an excess relative to the actual balance of the contract and an underfrom in the calculations.

## Recommendation

It is recommended to ensure that the state of the contract is always accurately reflected in the stored variables. This will enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

## ILT - Inconsistent Liquidity Tax

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L128 |
| **Status** | Unresolved |

### Description

The contract derives the `liquidityTax` as a portion of the sum of `buyTax` and `sellTax`. However, only one of these taxes applies to a given transaction. As a result, the contract may underestimate the amount to be swapped for liquidity.

```
uint256 liquidityTax = ((contractTokenBalance * liquidityAllocation) / (buyTax
+ sellTax));
```

### Recommendation

It is advised to monitor the current implementation to ensure that the swap amounts align with the intended design and that all available balances are utilized efficiently.

## MCM - Misleading Comment Messages

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L39 |
| **Status** | Unresolved |

### Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

```
uint256 public minimumHoldingForReflection = 250 * 10 ** 18; // 250,000
tokens
```

### Recommendation

The team is advised to carefully review the comment in order to reflect the actual implementation. To improve code readability, the team should use more specific and descriptive comment messages.

## MVN - Misleading Variables Naming

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L146 |
| **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. In this case, the `buyTax` is invoked for all taxed transactions, even if they are not buys.

```
uint256 transactionTax = (to == liquidityPool) ? sellTax : buyTax;
```

### 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** | CryptoChamps.sol#L317 |
| **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.

```
function excludeFromFees(
address account,
bool excluded
) external onlyOwner {
_isExcludedFromFees[account] = excluded;
}
```

### Recommendation

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

## PMRM - Potential Mocked Router Manipulation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L70 |
| **Status** | Unresolved |

### Description

The contract includes a method that allows the owner to modify the router address and create a new pair. While this feature provides flexibility, it introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

```
constructor(
address _uniswapRouter,
address _admin
) ERC20("Champ", "CCG") Ownable(_msgSender()) {
...
}
```

### Recommendation

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.

## PVC - Price Volatility Concern

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L105 |
| **Status** | Unresolved |

### Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapTokensAtAmount` 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.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
function changeSwapThreshold(uint256 _tokens) external onlyOwner {
require(_tokens > 0, "Minimum token value must be higher than
0");swapTokensAtAmount = _tokens;
}
```

### 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.

## RRA - Redundant Repeated Approvals

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L154,202 |
| **Status** | Unresolved |

### Description

The contract is designed to `approve` token transfers during the contract's operation by calling the _approve function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
function _swapTokensForWETH(
uint256 tokenAmount
) private nonReentrant returns (uint256) {
...
_approve(address(this), address(uniswapRouter), tokenAmount);
...
}
```

### Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

## UA - Unsanitized Allocation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L308 |
| **Status** | Unresolved |

### Description

The contract accepts user-provided variables without proper validation to ensure they form the proper shape. Specifically, it does not verify whether the values are non-zero or within acceptable limits. If excessively large values are provided, significant inconsistencies may arise within the contract.

```
function setTaxAllocations(
uint256 _liquidityAllocation,
uint256 _reflectionAllocation
) external onlyOwner {
liquidityAllocation = _liquidityAllocation;
reflectionAllocation = _reflectionAllocation;
emit TaxAllocationsUpdated(_liquidityAllocation,
_reflectionAllocation);
}
```

### Recommendation

It is recommended to implement proper checks to ensure that user-provided values fall within acceptable ranges, thereby maintaining the contract's consistency.

## ZD - Zero Division

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L259 |
| **Status** | Unresolved |

### Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

```
uint256 withoutLiquidity = (taxBalance * 100) /
reflectionAllocation;
```

### Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L44,91,98,105,271,292,293,300,309,310 |
| **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.

```
mapping(address => bool) public _isExcludedFromFees
address _addr
uint256 _tokens
address _receiver
uint256 _amount
uint256 _sellTax
uint256 _buyTax
uint256 _liquidityAllocation
uint256 _reflectionAllocation
```

## 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.

## L07 - Missing Events Arithmetic

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L107 |
| **Status** | Unresolved |

### Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
swapTokensAtAmount = _tokens
```

### Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

## L13 - Divide before Multiply Operation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L161,162 |
| **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.

```
uint256 expectedOut = (r1 * tokenAmount) / (r0 + tokenAmount)
uint256 minOut = (expectedOut * 95) / 100
```

### 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.

## L16 - Validate Variable Setters

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | CryptoChamps.sol#L215 |
| **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.

```
(bool success, ) = to.call{value: amount}("")
```

### Recommendation

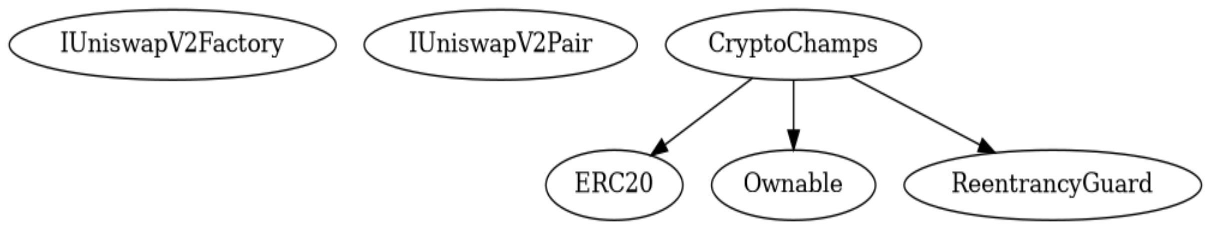
By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

## Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | Function Name | Visibility | Mutability | Modifiers |
| | | | | |
| CryptoChamps | Implementation | ERC20, Ownable, ReentrancyGuard | | |
| | | Public | ✓ | ERC20 Ownable |
| | changeAdmin | External | ✓ | onlyOwner |
| | changeMinimumHoldingForReflection | External | ✓ | onlyOwner |
| | changeSwapThreshold | External | ✓ | onlyOwner |
| | _createLiquidityPool | Internal | ✓ | |
| | _update | Internal | ✓ | |
| | _swapTokensForWETH | Private | ✓ | nonReentrant |
| | _wETHAmountAndPath | Private | | |
| | _addToLiquidity | Private | ✓ | |
| | emergencyWithdrawETH | External | ✓ | onlyOwner |
| | emergencyWithdrawToken | External | ✓ | onlyOwner |
| | _distributeReflections | Private | ✓ | |
| | calculateETHClaimable | Public | | - |
| | calculateETHWillBeClaimable | Public | | - |
| | claimReflections | External | ✓ | noFlashLoans nonReentrant |
| | _claimReflections | Internal | ✓ | |
| | claimRewardPointsWithCCG | External | ✓ | nonReentrant |

| | | | | |
|---|---|---|---|---|
| | setTaxes | External | ✓ | onlyOwner |
| | setTaxAllocations | External | ✓ | onlyOwner |
| | excludeFromFees | External | ✓ | onlyOwner |
| | | External | Payable | - |

## Inheritance Graphs



## Summary

CHAMP contract implements a token 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