Audit Report

CHAMP

March 2025

Repository :

https://github.com/sian0306/CryptoChamps-Audit-Repo/tree/main

Commit :

4fd668cb8e5a121eaca56e9e82a6f8c88f0afb90

Audited by © cyberscope

Analysis

Critical · Medium · Minor / Informative · Pass

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

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
● (Critical)	BUIBS	Balance Update Initiated Before Swap	Unresolved
● (Critical)	ISA	Inconsistent Swap Amount	Unresolved
● (Critical)	MRF	Missing Reentrancy Flag	Unresolved
● (Medium)	FLV	Flash Loan Vulnerability	Unresolved
● (Medium)	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
● (Medium)	TSI	Tokens Sufficiency Insurance	Unresolved
● (Minor / Informative)	CO	Code Optimization	Unresolved
● (Minor / Informative)	PRE	Potential Reentrance Exploit	Unresolved
● (Minor / Informative)	RCS	Redundant Conditional Statements	Unresolved
● (Minor / Informative)	RRA	Redundant Repeated Approvals	Unresolved
● (Minor / Informative)	RSD	Redundant Swap Duplication	Unresolved
● (Minor / Informative)	ZD	Zero Division	Unresolved
● (Minor / Informative)	L04	Conformance to Solidity Naming Conventions	Unresolved
● (Minor / Informative)	L13	Divide before Multiply Operation	Unresolved

●	L16	Validate Variable Setters	Unresolved
●	L20	Succeeded Transfer Check	Unresolved
●	L22	Potential Locked Ether	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

Repository	https://github.com/sian0306/deployment-repo/tree/main
Commit	4fd668cb8e5a121eaca56e9e82a6f8c88f0afb90
Badge Eligibility	Must Fix Criticals

Audit Updates

Initial Audit	09 Mar 2025 https://github.com/cyberscope-io/audits/blob/main/4-ccg/v1/audit.pdf
Corrected Phase 2	21 Mar 2025
Test Deploys	https://sepolia.etherscan.io/address/0x57ea706823C002cCeAdD74d82dEF8aB242AE30C8 https://sepolia.etherscan.io/address/0xc7ed61c0fE94919E3EFcbCD3cA2b092876f0d09b

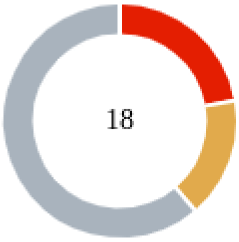
Source Files

Filename	SHA256
WETHHolder.sol	0b063b0b99880dfad33b23836010aa174276b12c05be13984b61129c837b36d5
CryptoChamps.sol	9eb589c41559dd91bf5b582a36729361e0a8ef3888ab4d45dba911c26c962693
interface/IWETHHolder.sol	304ec9f7a1975bd4c452b8f5d6ef0d4de3b8730195ebef7c8a8df775759a1690

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic, and propose potential optimizations. The contract does not adhere to best practices for interacting with other smart contracts and decentralized applications (dApps). Specifically, it fails to implement consistent cross-contract interaction patterns, leading to inconsistencies, failed transactions, and overall discrepancies. The development team is strongly advised to follow standardized methodologies for contract interoperability and decentralized application design, ensuring proper transaction flow, error handling, and state consistency. Adopting best practices will enhance the contract's reliability and prevent issues related to failed transactions and broken composability.

Findings Breakdown



Severity	Unresolved	Acknowledged	Resolved	Other
Critical	4	0	0	0
Medium	3	0	0	0
Minor / Informative	11	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	CryptoChamps.sol#L349
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 `pause` function. As a result, all transactions will be disabled for 6 hours. After that period transactions are automatically enabled. The owner may pause the transactions again after a cooldown period.

In addition, the owner may stop all transactions as described in findings `ZD`, `PLPI`. As a result the contract may operate as a honeypot.

```
function pause() public onlyOwner {
require(!paused(), "Already paused");
require(
block.timestamp >= lastUnpausedAt + PAUSE_COOLDOWN,
"Cooldown active: Cannot pause again yet"
);

pausedAt = block.timestamp;
_pause();
}

function unpause() public onlyOwner {
require(paused(), "Not paused");
_unpause();
lastUnpausedAt = block.timestamp;
}

function isPauseExpired() public view returns (bool) {
return paused() && (block.timestamp >= pausedAt +
MAX_PAUSE_DURATION);
}

modifier transferAllowed()  {
require(!paused() || isPauseExpired(), "Pausable: paused and time
limit not reached");
if (isPauseExpired()) {
_unpause(); // Auto unpause if expired
}
_;
}
```

Recommendation

The contract could embody a check for not allowing setting the `pause` duration less than a reasonable amount with a respectively reasonable cooldown period. In addition, the contract should prevent zero division to ensure consistency.

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.

BUIBS - Balance Update Initiated Before Swap

Criticality	Critical
Location	CryptoChamps.sol#L107
Status	Unresolved

Description

The contract is designed to perform a token swap after updating the respective balances. This sequence of operations can cause inconsistencies in the balances used for operations involving the router, which may result in the swap being halted. These inconsistencies disrupt the functionality of the swap, causing transactions to fail and affecting the overall liquidity operations of the contract.

```
super._update(from, address(this), taxAmount);
uint256 wethOutFromSwap = _swapTokensForWETH(tokensToSwap);
```

Recommendation

It is recommended to ensure that the swap is triggered only before the balances are updated. This will prevent inconsistencies in the balances used during router operations, ensuring the swap executes successfully and maintaining smooth token sale functionality. It is advisable to modify the code logic in accordance to finding `RSD` to resolve this issue.

ISA - Inconsistent Swap Amount

Criticality	Critical
Location	CryptoChamps.sol#L173
Status	Unresolved

Description

As part of the transfer flow, the contract swaps tokens for native currencies and provides liquidity with the exchanged funds. During execution, the contract uses the `getAmountsOut` function from the Uniswap router to calculate the necessary amount of native tokens for these operations. Initially, the contract swaps the accrued fees for native currencies and then uses the `getAmountsOut` method to estimate the amount of native tokens required for liquidity provision with the remaining half. This estimation might differ from the actual amount needed to complete the operation.

```
function _wETHAmountAndPath(
uint256 tokenAmount
) private view returns (uint256 amountOut, address[] memory path) {
require(tokenAmount > 0, "Token amount must be greater than zero");

path = new address[](2);
path[0] = address(this);
path[1] = uniswapRouter.WETH();

uint256[] memory amountsOut = uniswapRouter.getAmountsOut(
    tokenAmount,
    path
);
amountOut = amountsOut[1];
return (amountOut, path);
}
```

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.

MRF - Missing Reentrancy Flag

Criticality	Critical
Location	CryptoChamps.sol#L107
Status	Unresolved

Description

During the execution of `_swapTokensForWETH`, the Router handling the token swap, will attempt to transfer tokens from this contract. If the router is not exempt from fees, the swap will be triggered again, causing the entire transaction to revert. Therefore, a check is needed to ensure that a swap is only initiated when the contract is not already in the process of executing a swap. The use of nonReentrant modifier on `_swapTokensForWETH` is not sufficient as it will lead to revering transactions.

```
function _update(
address from,
address to,
uint256 amount
) internal override transferAllowed {
 ...
super._update(from, address(this), taxAmount);
uint256 wethOutFromSwap = _swapTokensForWETH(tokensToSwap);
...
}
```

```
modifier nonReentrant() {
_nonReentrantBefore();
_;
_nonReentrantAfter();
}

function _nonReentrantBefore() private {
// On the first call to nonReentrant, _status will be NOT_ENTERED
if (_status == ENTERED) {
revert ReentrancyGuardReentrantCall();
}

// Any calls to nonReentrant after this point will fail
_status = ENTERED;
}

function _nonReentrantAfter() private {
// By storing the original value once again, a refund is triggered (see
// https://eips.ethereum.org/EIPS/eip-2200)
_status = NOT_ENTERED;
}
```

Recommendation

To ensure transaction reliability and prevent failures, it is recommended to implement comprehensive validation checks that verify transaction states align with the specified requirements before execution. The team is advised to implement a control variable that prevents the eexecution of the `_swapTokensForWETH` when a swap is in process, while ensuring the remaining of the transaction can be executed without errors.

FLV - Flash Loan Vulnerability

Criticality	Medium
Location	CryptoChamps.sol#L218
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.

```
function calculateETHClaimable(
address holder
) public view returns (uint256) {
uint256 holderBalance = balanceOf(holder);
if (holderBalance < minimumHoldingForReflection) {
return 0;
}
uint256 totalSupplyExcludingBurned = totalSupply() -
balanceOf(address(0));
uint256 reflectionShare = (holderBalance *
totalReflectionsAccumulated) / 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#L155,190
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 returns
(uint256) {
(uint256 amountOut, address[] memory path) =
_wETHAmountAndPath(tokenAmount);
_approve(address(this), address(uniswapRouter), tokenAmount);
uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens
(
tokenAmount,
0,
path,
address(_wethHolder),
block.timestamp
);
_wethHolder.transferTokens(uniswapRouter.WETH(), address(this));
return amountOut;
}
```

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);
}
```

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.

CO - Code Optimization

Criticality	Minor / Informative
Location	CryptoChamps.sol#L155
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations. In particular, the `_swapTokensForWETH` method transfers the received tokens to the `_wethHolder` which then forwards its balance to the current contract.

```
function _swapTokensForWETH(uint256 tokenAmount) private returns
(uint256) {
(uint256 amountOut, address[] memory path) = _wETHAmountAndPath(
tokenAmount
);
_approve(address(this), address(uniswapRouter), tokenAmount);
uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens
(
tokenAmount,
0,
path,
address(_wethHolder),
block.timestamp
);
_wethHolder.transferTokens(uniswapRouter.WETH(), address(this));
return amountOut;
}
```

```
function transferTokens(address token, address to) external
onlyOwner {
require(to != address(0), "Invalid recipient address");
uint256 amount = IERC20(token).balanceOf(address(this));
require(amount > 0, "Invalid transfer amount");

(token).transfer(to, amount);
emit TokenWithdrawn(token, to, amount);
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	WETHHolder.sol#L28
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the payable transfer method. The recipient could be a malicious contract or a vulnerable contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract's funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

```
function withdrawEther(
address payable to,
uint256 amount
) external onlyOwner {
require(to != address(0), "Invalid recipient address");
require(amount > 0, "Invalid withdrawal amount");
require(address(this).balance >= amount, "Insufficient Ether
balance");

to.transfer(amount);
emit EtherWithdrawn(to, amount);
}
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.

- Proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	CryptoChamps.sol#L140
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that are always satisfied are unnecessary and lead to larger code size, increased memory usage, and slower execution times.

```
if (liquidityTax > 0)
wethUsedInLiquidity = _addToLiquidity(liquidityHalf);
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage. Simplifying such statements makes the code more readable and improves its overall performance.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	CryptoChamps.sol#L161,196
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 returns
(uint256) {
(uint256 amountOut, address[] memory path) =
_wETHAmountAndPath(tokenAmount);
_approve(address(this), address(uniswapRouter), tokenAmount);
uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens
(
tokenAmount,
0,
path,
address(_wethHolder),
block.timestamp
);
_wethHolder.transferTokens(uniswapRouter.WETH(), address(this));
return amountOut;
}
```

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.

RSD - Redundant Swap Duplication

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

Description

The contract executes a swap method for all transfers of the users. This redundancy introduces unnecessary complexity and increases dramatically the gas consumption. By consolidating accrued fees and performing a single swap method, the contract could achieve better code readability, reduce gas costs, and improve overall efficiency.

```
function _update(
address from,
address to,
uint256 amount
) internal override transferAllowed {
...
uint256 liquidityTax = ((taxAmount * liquidityAllocation) /
transactionTax);
uint256 reflectionTax = taxAmount - liquidityTax;
uint256 liquidityHalf = liquidityTax / 2;
uint256 swapHalf = liquidityTax - liquidityHalf;
uint256 tokensToSwap = swapHalf + reflectionTax;
super._update(from, address(this), taxAmount);
uint256 wethOutFromSwap = _swapTokensForWETH(tokensToSwap);
uint256 wethUsedInLiquidity = 0;

// Handle taxes
if (liquidityTax > 0)
wethUsedInLiquidity = _addToLiquidity(liquidityHalf);
if (reflectionTax > 0)
_distributeReflections(wethOutFromSwap - wethUsedInLiquidity);
...
}
```

Recommendation

A more optimized approach could be adopted to perform the token swap operation once a minimum amount of tokens is accumulated in the contract, eliminating the need for multiple swaps.

ZD - Zero Division

Criticality	Minor / Informative
Location	CryptoChamps.sol#L107,282
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.

```
function _update(
address from,
address to,
uint256 amount
) internal override transferAllowed {
...
uint256 transactionTax = (to == liquidityPool) ? sellTax : buyTax;
...
uint256 liquidityTax = ((taxAmount * liquidityAllocation) /
transactionTax);
...
}
```

```
function setTaxes(uint256 _buyTax, uint256 _sellTax) external
onlyOwner {
    require(_buyTax <= 10 && _sellTax <= 10, "Tax cannot exceed
10%");
    buyTax = _buyTax;
    sellTax = _sellTax;
    emit TaxesUpdated(_buyTax, _sellTax);
}
```

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#L86,93,233,274,275,282,290,291
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 _addr
uint256 _tokens
address _receiver
uint256 _amount
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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	CryptoChamps.sol#L120,129
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 taxAmount = (amount * transactionTax) / 100
uint256 liquidityTax = ((taxAmount * liquidityAllocation) /
transactionTax)
```

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#L83
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.

```
admin = _admin
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	WETHHolder.sol#L35
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(token).transfer(to, amount)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

L22 - Potential Locked Ether

Criticality	Minor / Informative
Location	CryptoChamps.sol#L342
Status	Unresolved

Description

The contract contains Ether that has been placed into a Solidity contract and is unable to be transferred. Thus, it is impossible to access the locked Ether. This may produce a financial loss for the users that have called the payable method.

```
fallback() external payable {}
```

Recommendation

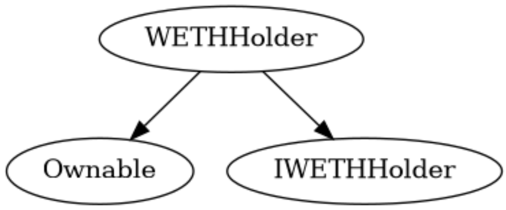
The team is advised to either remove the payable method or add a withdraw functionality. It is important to carefully consider the risks and potential issues associated with locked Ether.

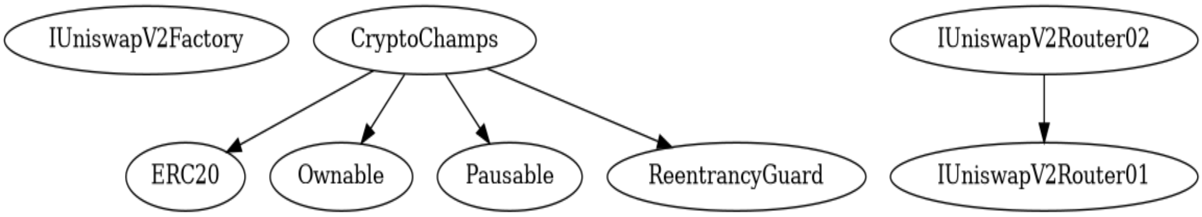
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
WETHHolder	Implementation	Ownable, IWETHHolder		
		External	Payable	-
		Public	✓	Ownable
	transferTokens	External	✓	onlyOwner
	withdrawEther	External	✓	onlyOwner
	getTokenBalance	External		-
CryptoChamps	Implementation	ERC20, Ownable, Pausable, ReentrancyGuard		
		Public	✓	ERC20 Ownable
	changeAdmin	External	✓	onlyOwner
	changeMinimumHoldingForReflection	External	✓	onlyOwner
	_createLiquidityPool	Internal	✓	
	_update	Internal	✓	transferAllowed
	_swapTokensForWETH	Private	✓	nonReentrant
	_wETHAmountAndPath	Private		
	_addToLiquidity	Private	✓	
	_distributeReflections	Private	✓	

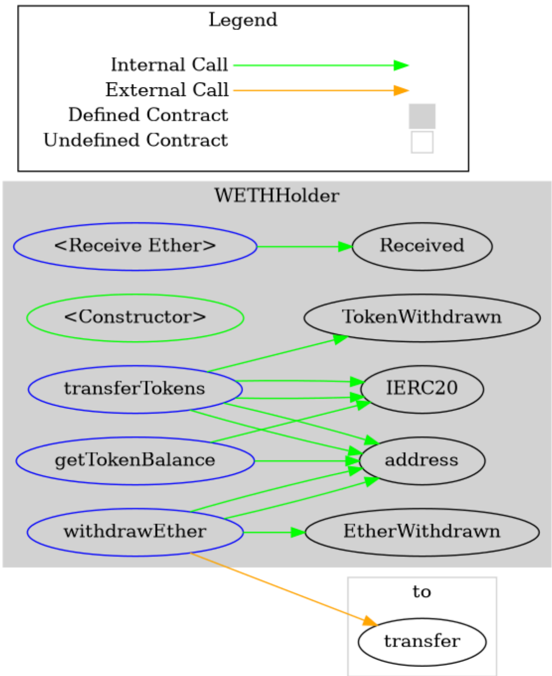
	calculateETHClaimable	Public		-
	claimReflections	External	✓	nonReentrant
	_claimReflections	Private	✓	nonReentrant
	claimRewardPointsWithCHP	External	✓	nonReentrant
	setTaxes	External	✓	onlyOwner
	setTaxAllocations	External	✓	onlyOwner
	pause	Public	✓	onlyOwner
	unpause	Public	✓	onlyOwner
	isPauseExpired	Public		-
	excludeFromFees	External	✓	onlyOwner
		External	Payable	-
		External	Payable	-
	onERC20Receive	External	✓	-
IWETHHolder	Interface			
	transferTokens	External	✓	-
	withdrawEther	External	✓	-
	getTokenBalance	External		-

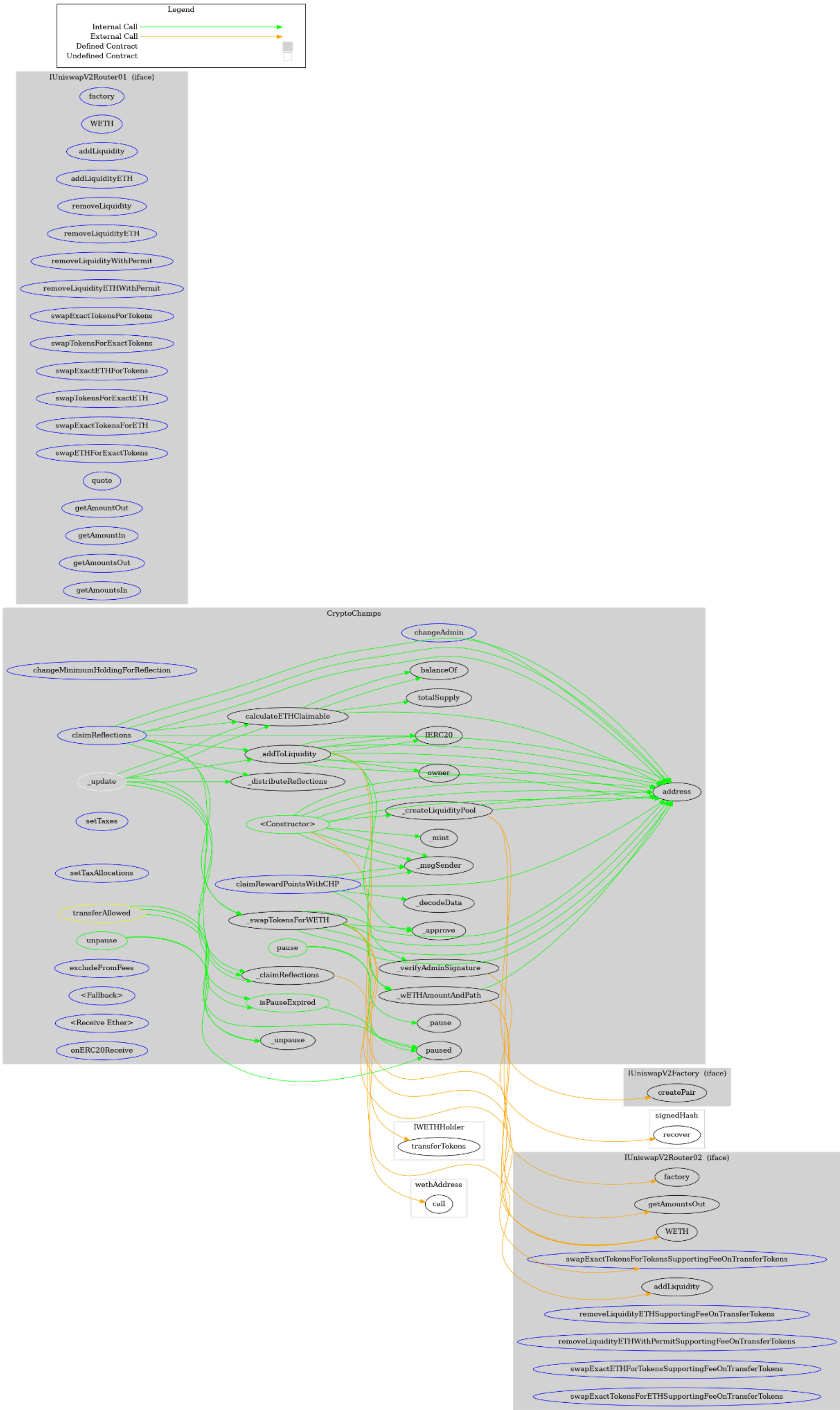
Inheritance Graphs





Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
WETHHolder
<Receive Ether>
Received
<Constructor>
TokenWithdrawn
transferTokens
IERC20
getTokenBalance
address
withdrawEther
EtherWithdrawn
to
transfer


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV2Router01 (iface)
factory
WETH
addLiquidity
addLiquidityETH
removeLiquidity
removeLiquidityETH
removeLiquidityWithPermit
removeLiquidityETHWithPermit
swapExactTokensForTokens
swapTokensForExactTokens
swapExactETHForTokens
swapTokensForExactETH
swapExactTokensForETH
swapETHForExactTokens
quote
getAmountOut
getAmountIn
getAmountsOut
getAmountsIn
CryptoChamps
changeAdmin
changeMinimumHoldingForReflection
balanceOf
totalSupply
calculateETHClaimable
claimReflections
IERC20
addToLiquidity
_update
_distributeReflections
owner
address
_createLiquidityPool
setTaxes
<Constructor>
mint
setTaxAllocations
_msgSender
claimRewardPointsWithCHP
_decodeData
transferAllowed
swapTokensForWETH
_approve
unpause
pause
_verifyAdminSignature
excludeFromFees
_claimReflections
<Fallback>
_wETHAmountAndPath
isPauseExpired
<Receive Ether>
pause
onERC20Receive
_unpause
paused
IUniswapV2Factory (iface)
createPair
signedHash
recover
IWETHHolder
transferTokens
IUniswapV2Router02 (iface)
factory
wethAddress
call
getAmountsOut
WETH
swapExactTokensForTokensSupportingFeeOnTransferTokens
addLiquidity
removeLiquidityETHSupportingFeeOnTransferTokens
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens

Summary

CHAMP contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. The audit analysis reported a number of high severity concerns. The team is advised to take these findings into consideration to improve the overall security of the contracts.

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