Cyberscope

Audit Report

CHAMP

March 2025

Repository :

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

Commit :

54f37431b314fb6cfc1ccfe319310b9bbee8a428

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
●	BUIBS	Balance Update Initiated Before Swap	Unresolved
●	ISA	Inconsistent Swap Amount	Unresolved
●	MRF	Missing Reentrancy Flag	Unresolved
●	FLV	Flash Loan Vulnerability	Unresolved
●	INU	Inconsistent Nonce Update	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	CO	Code Optimization	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	ISV	Ineffective Signature Validation	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PRE	Potential Reentrance Exploit	Unresolved
●	RCS	Redundant Conditional Statements	Unresolved

●	RRA	Redundant Repeated Approvals	Unresolved
●	RSD	Redundant Swap Duplication	Unresolved
●	SVMC	Signature Validation Missing ChainID	Unresolved
●	ZD	Zero Division	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	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
Test Deploys	https://sepolia.etherscan.io/address/0xb6c4aA3ddDA8968E9A2C897714851a26c56A94b1 https://sepolia.etherscan.io/address/0xe71b0dF27ecD99058d10b378165eA55D676E1704

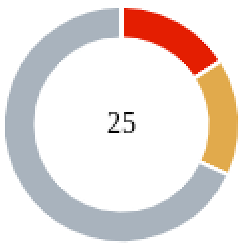
Source Files

Filename	SHA256
WETHHolder.sol	0b063b0b99880dfad33b23836010aa174276b12c05be13984b61129c837b36d5
CryptoChamps.sol	da518b6b8b909ad79d076caad3522bcc6f02c844d68a68ac39d83caa40f478a1
interface/IWETHHolder.sol	52f0cb914c94c870f4012f855de359e087fdebf3d14a1aa560dce796aa799c6d

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



●	Critical	4
●	Medium	4
●	Minor / Informative	17

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	4	0	0	0
●	Medium	4	0	0	0
●	Minor / Informative	17	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. Afer 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 while ensuring transfers will be processed irrespectively of the presence of liquidity.

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.

ISA - Inconsistent Swap Amount

Criticality	Critical
Location	CryptoChamps.sol#L170
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.

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

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.

FLV - Flash Loan Vulnerability

Criticality	Medium
Location	CryptoChamps.sol#L217
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.

INU - Inconsistent Nonce Update

Criticality	Medium
Location	CryptoChamps.sol#L273
Status	Unresolved

Description

The `claimRewardPointsWithCHP` function can only be called by the admin account. The `_decodeData` method retrieves the nonce from the signature, which is then compared to the value stored in `userNonce` for the admin. This is an attempt to ensure the uniqueness of the accepted signatures. However, the contract increments the nonce for the `userAddress` account and not that of the admin. Since the `userAddress` may differ from the admin, an inconsistency arises between the expected nonce for the admin and the increase of nonce values from past signatures.

```
function claimRewardPointsWithCHP(
bytes memory encryptedData,
bytes memory signature
) external nonReentrant {
...
userNonce[userAddress]++;
...
}
```

Recommendation

It is advisable to properly update the nonce in the contract's state to reflect the signatures signed by the admin. This will ensure the uniqueness of the signatures and enhance trust in the system.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Medium
Location	CryptoChamps.sol#L160,195
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#L156
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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	CryptoChamps.sol#L77,80
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
wethAddress
liquidityPool
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

ISV - Ineffective Signature Validation

Criticality	Minor / Informative
Location	CryptoChamps.sol#L273
Status	Unresolved

Description

The `claimRewardPointsWithCHP` function relies on signature validation to process token transfers from the admin's account to a specified address. However, since the function can only be called by the admin, who is also the entity that signs the signature, the presence of a signature validation mechanism is ineffective. The admin could process the transfer without the requirement of a signature.

```
function claimRewardPointsWithCHP(
bytes memory encryptedData,
bytes memory signature
) external nonReentrant {
(
...
require(_msgSender() == admin, "Only Admin Can Call");
...
require(
_verifyAdminSignature(
    userAddress,
    amount,
    timestamp,
    nonces,
    signature
    ),
"Invalid admin signature"
);
...
}
```

Recommendation

The current implementation could be improved by allowing the admin to directly call the relevant methods. Alternatively, the provided signature may be signed by a separate entity than the caller. The team might consider implementing a multisignature mechanism to enhance the security of the admin account.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
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 particular, the `buyTax` is assigned to the `taxRate` even when the transaction is not a `buy`.

```
uint256 taxRate = (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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Status	Unresolved

Description

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

```
function changeAdmin(address _addr) external onlyOwner {
require(_addr != address(0), "Zero address");
admin = _addr;
}

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

Recommendation

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

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#L141
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#L160,195
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) / taxRate;
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.

SVMC - Signature Validation Missing ChainID

Criticality	Minor / Informative
Location	CryptoChamps.sol#L313
Status	Unresolved

Description

The contract's `_verifyAdminSignature` function is designed to validate off-chain signatures for operations involving token transfers between addresses. However, the function does not include the `chainId` as part of the parameters in the signature verification process. While the use of a nonce can prevent replay attacks within the same network by ensuring each signature is unique for a particular transaction, it does not safeguard against replay attacks across different networks. Without the inclusion of `chainId`, a legitimate signature on one blockchain could be maliciously reused on another chain, potentially resulting in unintended or unauthorized token transfers, thus exposing the contract to cross-network vulnerabilities.

```
function _verifyAdminSignature(
address userAddress,
uint256 amount,
uint256 timestamp,
uint256 nonces,
bytes memory signature
) internal view returns (bool) {
bytes32 messageHash = keccak256(
abi.encodePacked(userAddress, amount, timestamp, nonces)
);
bytes32 signedHash = keccak256(
abi.encodePacked("\x19Ethereum Signed Message:\n32", messageHash)
);
return signedHash.recover(signature) == admin;
}
```

Recommendation

It is recommended to incorporate the `chainId` in the signature verification process by including it in the parameters hashed during the signature construction. By doing so, the signatures will be explicitly tied to a specific network, effectively preventing them from being reused across different chains.

ZD - Zero Division

Criticality	Minor / Informative
Location	CryptoChamps.sol#L107,329
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 taxRate = (to == liquidityPool) ? sellTax : buyTax;
...
uint256 liquidityTax = (taxAmount * liquidityAllocation) / taxRate;
...
}
```

```
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#L87,93,230,326,334,335
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 _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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	CryptoChamps.sol#L43
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
mapping(address => uint256) private lastClaimedIndex
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

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 * taxRate) / 100
uint256 liquidityTax = (taxAmount * liquidityAllocation) / taxRate
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	interface/IWETHHolder.sol#L2
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 ^0.8.0;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	WETHHolder.sol#L22
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

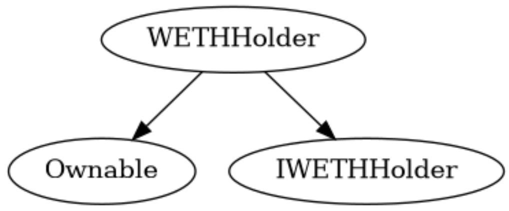
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

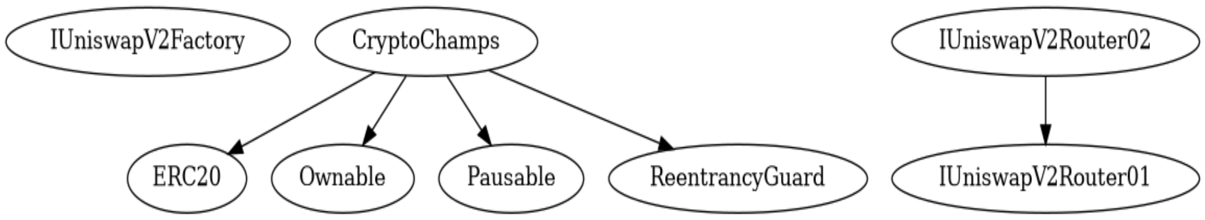
Functions Analysis

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	✓	
	_wETHAmountAndPath	Private		
	_addToLiquidity	Private	✓	
	_distributeReflections	Private	✓	

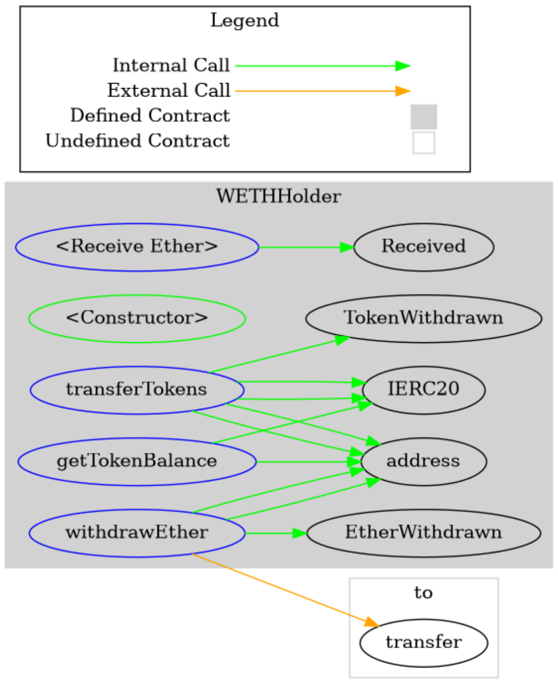
	calculateETHClaimable	Public		-
	claimReflections	External	✓	-
	_claimReflections	Private	✓	nonReentrant
	claimRewardPointsWithCHP	External	✓	nonReentrant
	_decodeData	Internal		
	_verifyAdminSignature	Internal		
	setTaxes	External	✓	onlyOwner
	setTaxAllocations	External	✓	onlyOwner
	pause	Public	✓	onlyOwner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

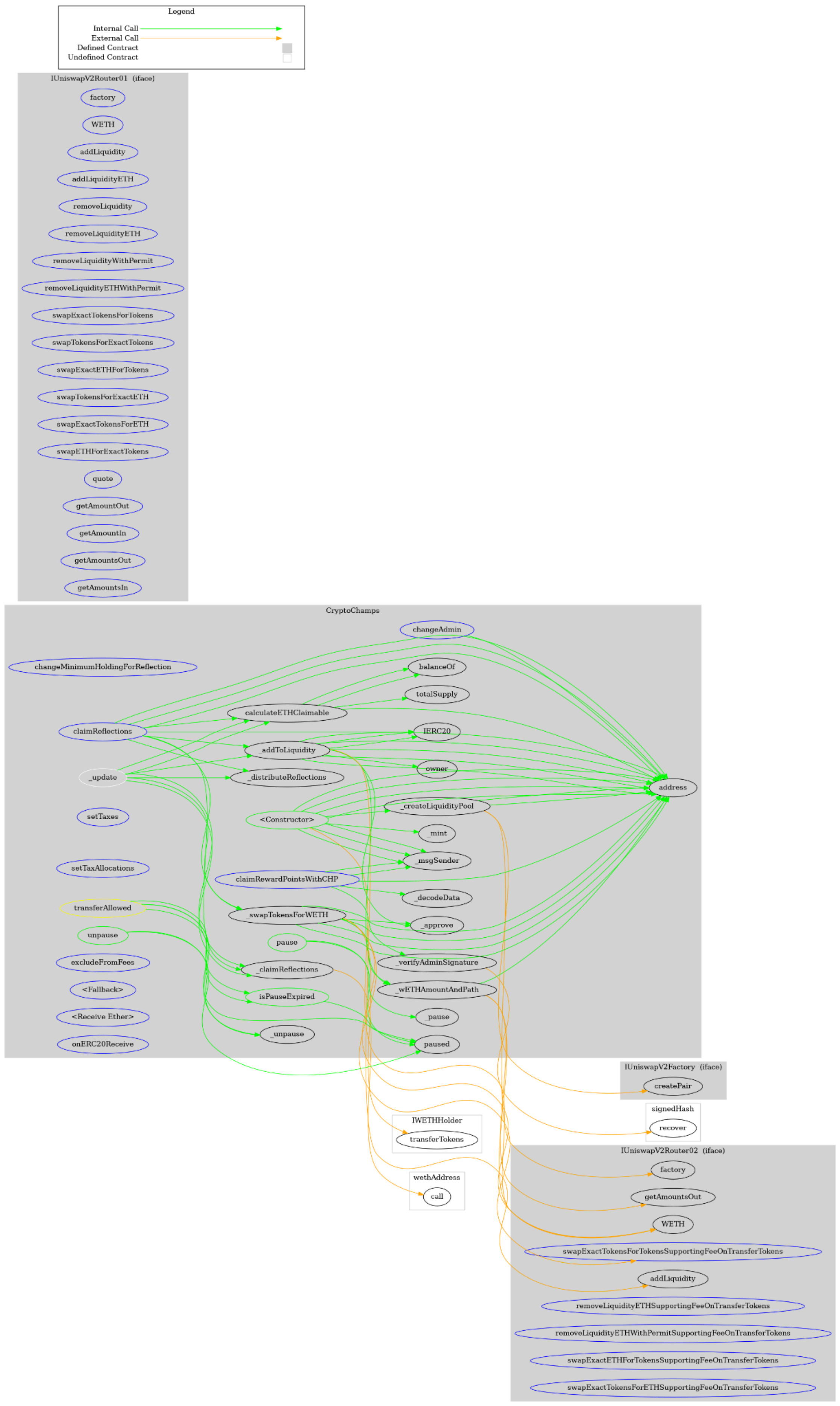
Inheritance Graphs





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
owner
_update
_distributeReflections
address
_createLiquidityPool
setTaxes
<Constructor>
mint
_msgSender
setTaxAllocations
claimRewardPointsWithCHP
_decodeData
transferAllowed
swapTokensForWETH
_approve
unpause
pause
excludeFromFees
_verifyAdminSignature
_claimReflections
<Fallback>
_wETHAmountAndPath
isPauseExpired
<Receive Ether>
pause
_unpause
onERC20Receive
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

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