



# SECURITY ASSESSMENT **cetwifweed** TOKEN

April 6, 2024

Audit Status: Fail














# RISK ANALYSIS | cetwifweed.

## ■ Classifications of Manual Risk Results

Classification	Description
 Critical	Danger or Potential Problems.
 High	Be Careful or Fail test.
 Medium	Improve is needed.
 Low	Pass, Not-Detected or Safe Item.
 Informational	Function Detected

## ■ Manual Code Review Risk Results

Contract Security	Description
 Buy Tax	25%
 Sale Tax	25%
 Cannot Buy	Pass
 Cannot Sale	Pass
 Max Tax	25%
 Modify Tax	Yes
 Fee Check	Pass
 Is Honeypot?	Not Detected
 Trading Cooldown	Not Detected
 Enable Trade?	false
 Pause Transfer?	Not Detected

Contract Security	Description
● Max Tx?	Fail
● Is Anti Whale?	Detected
● Is Anti Bot?	Detected
● Is Blacklist?	Not Detected
● Blacklist Check	Pass
● is Whitelist?	Detected
● Can Mint?	Pass
● Is Proxy?	Not Detected
● Can Take Ownership?	Not Detected
● Hidden Owner?	Not Detected
● Owner	0x8C49A61F7Fe943da6d921719DFc4ABb56c2b8877
● Self Destruct?	Not Detected
● External Call?	Detected
● Other?	Not Detected
● Holders	3
● Audit Confidence	Very Low
● Authority Check	Pass
● Freeze Check	Pass

The summary section reveals the strengths and weaknesses identified during the assessment, including any vulnerabilities or potential risks that may exist. It serves as a valuable snapshot of the overall security status of the audited project. However, it is highly recommended to read the entire security assessment report for a comprehensive understanding of the findings. The full report provides detailed insights into the assessment process, methodology, and specific recommendations for addressing the identified issues.

CFG Ninja Verified on April 6, 2024

cetwifweed



## Executive Summary

TYPES

DeFi

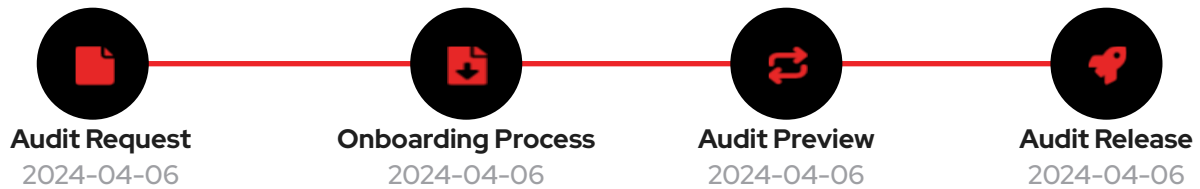
ECOSYSTEM

BASE

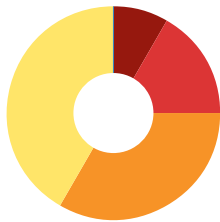
LANGUAGE

Solidity

## Timeline



## Vulnerability Summary



Total Findings

0

Resolved

12

Pending

0

Unresolved

### 1 Critical

0 Resolved, 1 Pending

Critical risks are the most severe and can have a significant impact on the smart contracts functionality, security, or the entire system. These vulnerabilities can lead to the loss of user funds, unauthorized access, or complete system compromise.

### 2 High

0 Resolved, 2 Pending

High-risk vulnerabilities have the potential to cause significant harm to the smart contract or the system. While not as severe as critical risks, they can still result in financial losses, data breaches, or denial of service attacks.

### 4 Medium

0 Resolved, 4 Pending

Medium-risk vulnerabilities pose a moderate level of risk to the smart contracts security and functionality. They may not have an immediate and severe impact but can still lead to potential issues if exploited. These risks should be addressed to ensure the contracts overall security.

### 5 Low

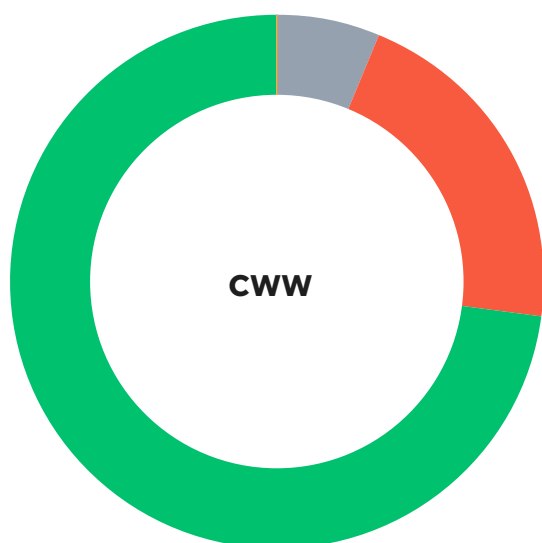
0 Resolved, 5 Pending

Low-risk vulnerabilities have a minimal impact on the smart contracts security and functionality. They may not pose a significant threat, but it is still advisable to address them to maintain a robust security posture.

### 0 Informational

Informational risks are not actual vulnerabilities but provide useful information about potential improvements or best practices. These findings may include suggestions for code optimizations, documentation enhancements, or other non-critical areas for improvement.

## Token Distribution



### Burn

Burned amount send to the deadWallet.

6%

### Liquidity

Liquidity tokens are split from sale into the pool.

20%

### Presale

Tokens allocated for the sale.

70%

### Staking - Listing

Ecosystem

0%

### Team and Advisors

Teams

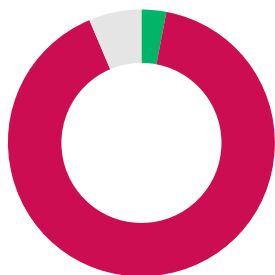
0%

### Reserves

Community

0%

## Total Unlock Progress



Unlocked	3000000	3%
Total Locked	90615000	90.615%
Untracked	6385000	6.385%

# PROJECT OVERVIEW | cetwifweed.

## Token Summary

Parameter	Result
Address	0xfA31a1298204b28A90a211cD7c5d78cEf786B23B
Name	cetwifweed
Token Tracker	cetwifweed (cww)
Decimals	18
Supply	100,000,000
Platform	BASE
Compiler	v0.8.10+commit.fc410830
Contract Name	CWW
Optimization	Yes with 200 runs
LicenseType	MIT
Language	Solidity
Codebase	<a href="https://basescan.org/address/0xfA31a1298204b28A90a211cD7c5d78cEf786B23B#code">https://basescan.org/ address/0xfA31a1298204b28A90a211cD7c5d78cEf786B23B#code</a>

## ■ Main Contract Assessed

Name	Contract	Live
cetwifweed	0xfA31a1298204b28A90a211cD7c5d78cEf786B23B	Yes

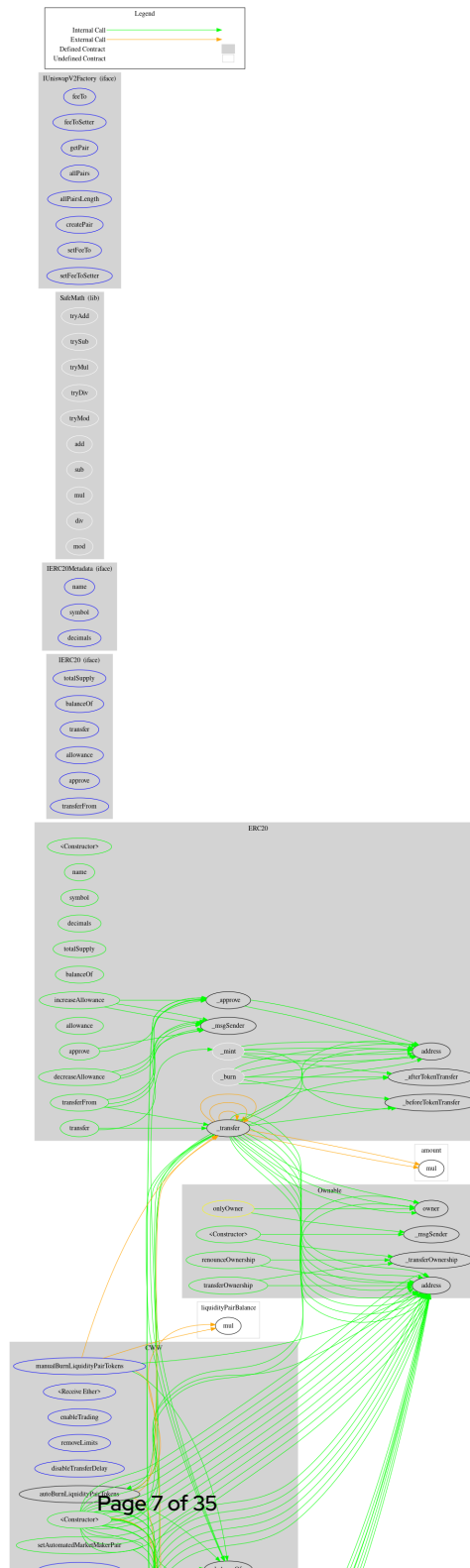
## ■ TestNet Contract Was Not Assessed

## ■ Solidity Code Provided

SolidID	File Sha-1	FileName
CWW	9101e3d6ad3feaf808678dfb522896e0638a5a2d	CWW.sol

## Call Graph

The Smart Contract Graph is a visual representation of the interconnectedness and relationships between smart contracts within a blockchain network. It provides a comprehensive view of the interactions and dependencies between different smart contracts, allowing developers and users to analyze and understand the flow of data and transactions within the network. The Smart Contract Graph enables better transparency, security, and efficiency in decentralized applications by facilitating the identification of potential vulnerabilities, optimizing contract execution, and enhancing overall network performance.





## Smart Contract Vulnerability Checks

The Smart Contract Weakness Classification Registry (SWC Registry) is an implementation of the weakness classification scheme proposed in EIP-1470. It is loosely aligned to the terminologies and structure used in the Common Weakness Enumeration (CWE) while overlaying a wide range of weakness variants that are specific to smart contracts.

ID	Severity	Name	File	location
SWC-100	Pass	Function Default Visibility	CWW.sol	L: 0 C: 0
SWC-101	Pass	Integer Overflow and Underflow.	CWW.sol	L: 0 C: 0
SWC-102	Pass	Outdated Compiler Version file.	CWW.sol	L: 0 C: 0
SWC-103	Low	A floating pragma is set.	CWW.sol	L: 12 C: 0
SWC-104	Pass	Unchecked Call Return Value.	CWW.sol	L: 0 C: 0
SWC-105	Pass	Unprotected Ether Withdrawal.	CWW.sol	L: 0 C: 0
SWC-106	Pass	Unprotected SELFDESTRUCT Instruction	CWW.sol	L: 0 C: 0
SWC-107	Pass	Read of persistent state following external call.	CWW.sol	L: 0 C: 0
SWC-108	Pass	State variable visibility is not set..	CWW.sol	L: 0 C: 0
SWC-109	Pass	Uninitialized Storage Pointer.	CWW.sol	L: 0 C: 0
SWC-110	Pass	Assert Violation.	CWW.sol	L: 0 C: 0
SWC-111	Pass	Use of Deprecated Solidity Functions.	CWW.sol	L: 0 C: 0
SWC-112	Pass	Delegate Call to Untrusted Callee.	CWW.sol	L: 0 C: 0
SWC-113	Pass	Multiple calls are executed in the same transaction.	CWW.sol	L: 0 C: 0
SWC-114	Pass	Transaction Order Dependence.	CWW.sol	L: 0 C: 0
SWC-115	Low	Authorization through tx.origin.	CWW.sol	L: 857 C: 57, L: 861 C: 53
SWC-116	Pass	A control flow decision is made based on The block.timestamp environment variable.	CWW.sol	L: 0 C: 0
SWC-117	Pass	Signature Malleability.	CWW.sol	L: 0 C: 0

ID	Severity	Name	File	location
SWC-118	Pass	Incorrect Constructor Name.	CWW.sol	L: 0 C: 0
SWC-119	Pass	Shadowing State Variables.	CWW.sol	L: 0 C: 0
SWC-120	Low	Potential use of block.number as source of randomness.	CWW.sol	L: 858 C: 32, L: 861 C: 66
SWC-121	Pass	Missing Protection against Signature Replay Attacks.	CWW.sol	L: 0 C: 0
SWC-122	Pass	Lack of Proper Signature Verification.	CWW.sol	L: 0 C: 0
SWC-123	Pass	Requirement Violation.	CWW.sol	L: 0 C: 0
SWC-124	Pass	Write to Arbitrary Storage Location.	CWW.sol	L: 0 C: 0
SWC-125	Pass	Incorrect Inheritance Order.	CWW.sol	L: 0 C: 0
SWC-126	Pass	Insufficient Gas Griefing.	CWW.sol	L: 0 C: 0
SWC-127	Pass	Arbitrary Jump with Function Type Variable.	CWW.sol	L: 0 C: 0
SWC-128	Pass	DoS With Block Gas Limit.	CWW.sol	L: 0 C: 0
SWC-129	Pass	Typographical Error.	CWW.sol	L: 0 C: 0
SWC-130	Pass	Right-To-Left-Override control character (U+202E).	CWW.sol	L: 0 C: 0
SWC-131	Pass	Presence of unused variables.	CWW.sol	L: 0 C: 0
SWC-132	Pass	Unexpected Ether balance.	CWW.sol	L: 0 C: 0
SWC-133	Pass	Hash Collisions with Multiple Variable Length Arguments.	CWW.sol	L: 0 C: 0
SWC-134	Pass	Message call with hardcoded gas amount.	CWW.sol	L: 0 C: 0
SWC-135	Pass	Code With No Effects (Irrelevant/ Dead Code).	CWW.sol	L: 0 C: 0
SWC-136	Pass	Unencrypted Private Data On-Chain.	CWW.sol	L: 0 C: 0

We scan the contract for additional security issues using MYTHX and industry-standard security scanning tools.

## Smart Contract Vulnerability Details | SWC-103 – Floating Pragma.

### CWE-664: Improper Control of a Resource Through its Lifetime.

#### References:

#### Description:

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

#### Remediation:

Lock the pragma version and also consider known bugs (<https://github.com/ethereum/solidity/releases>) for the compiler version that is chosen.

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package.

Otherwise, the developer would need to manually update the pragma in order to compile locally.

#### References:

Ethereum Smart Contract Best Practices – Lock pragmas to specific compiler version.

## Smart Contract Vulnerability Details | SWC-115 - Authorization through tx.origin.

### CWE-477: Use of Obsolete Function

#### Description:

tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.

#### Remediation:

tx.origin should not be used for authorization. Use msg.sender instead.

#### References:

Solidity Documentation - tx.origin

Ethereum Smart Contract Best Practices - Avoid using tx.origin

SigmaPrime - Visibility.

## Smart Contract Vulnerability Details | SWC-120 - Weak Sources of Randomness from Chain Attributes.

### CWE-330: Use of Insufficiently Random Values

#### Description:

Solidity allows for ambiguous naming of state variables when inheritance is used. Contract A with a variable x could inherit contract B that also has a state variable x defined. This would result in two separate versions of x, one of them being accessed from contract A and the other one from contract B. In more complex contract systems this condition could go unnoticed and subsequently lead to security issues.

Shadowing state variables can also occur within a single contract when there are multiple definitions on the contract and function level.

#### Remediation:

Using commitment scheme, e.g. RANDAO. Using external sources of randomness via oracles, e.g. Oraclize. Note that this approach requires trusting in oracle, thus it may be reasonable to use multiple oracles. Using Bitcoin block hashes, as they are more expensive to mine.

#### References:

How can I securely generate a random number in my smart contract?)






When can BLOCKHASH be safely used for a random number? When would it be unsafe?

The Run smart contract.

## TECHNICAL FINDINGS | cetwifweed.


Smart contract security audits classify risks into several categories: Critical, High, Medium, Low, and Informational. These classifications help assess the severity and potential impact of vulnerabilities found in smart contracts.

### Classification of Risk

Severity	Description
 Critical	Critical risks are the most severe and can have a significant impact on the smart contracts functionality, security, or the entire system. These vulnerabilities can lead to the loss of user funds, unauthorized access, or complete system compromise.
 High	High-risk vulnerabilities have the potential to cause significant harm to the smart contract or the system. While not as severe as critical risks, they can still result in financial losses, data breaches, or denial of service attacks.
 Medium	Medium-risk vulnerabilities pose a moderate level of risk to the smart contracts security and functionality. They may not have an immediate and severe impact but can still lead to potential issues if exploited. These risks should be addressed to ensure the contracts overall security.
 Low	Low-risk vulnerabilities have a minimal impact on the smart contracts security and functionality. They may not pose a significant threat, but it is still advisable to address them to maintain a robust security posture.
 Informational	Informational risks are not actual vulnerabilities but provide useful information about potential improvements or best practices. These findings may include suggestions for code optimizations, documentation enhancements, or other non-critical areas for improvement.

By categorizing risks into these classifications, smart contract security audits can prioritize the resolution of critical and high-risk vulnerabilities to ensure the contract's overall security and protect user funds and data.

## cww-01 | Potential Sandwich Attacks.

Category	Severity	Location	Status
Security	<span>Medium</span>	CWW.sol: L:994, C:14	 Detected

### Description

A sandwich attack might happen when an attacker observes a transaction swapping tokens or adding liquidity without setting restrictions on slippage or minimum output amount. The attacker can manipulate the exchange rate by frontrunning (before the transaction being attacked) a transaction to purchase one of the assets and make profits by back running (after the transaction being attacked) a transaction to sell the asset. The following functions are called without setting restrictions on slippage or minimum output amount, so transactions triggering these functions are vulnerable to sandwich attacks, especially when the input amount is large:

- swapExactTokensForETHSupportingFeeOnTransferTokens()

### Recommendation



We recommend setting reasonable minimum output amounts, instead of 0, based on token prices when calling the aforementioned functions.

### Mitigation

### References:

What Are Sandwich Attacks in DeFi – and How Can You Avoid Them?.

## cww-03 | Lack of Input Validation.

Category	Severity	Location	Status
Volatile Code	 Low	CWW.sol: L: 779-813 C: 14	 Detected

### Description

The given input is missing the check for the non-zero address.

The given input is missing the check for the onlyOwners need to be revisited for require..

### Recommendation

We advise the client to add the check for the passed-in values to prevent unexpected errors as below:

```
...
require(receiver != address(0), "Receiver is the zero address");
...
...
require(value X limitation, "Your not able to do this function");
...
```

We also recommend customer to review the following function that is missing a required validation. onlyOwners need to be revisited for require..



### Mitigation

#### References:

Zero Address check. The danger!!!



## cww-05 | Missing Event Emission.

Category	Severity	Location	Status
Volatile Code	 Low	CWW.sol: L: 779-813 C: 14	 Detected

### Description

Detected missing events for critical arithmetic parameters. There are functions that have no event emitted, so it is difficult to track off-chain changes. The linked code does not create an event for the transfer.

### Recommendation



Emit an event for critical parameter changes. It is recommended emitting events for the sensitive functions that are controlled by centralization roles.

### Mitigation

### References:

Understanding Events in Smart Contracts

## cww-06 | Conformance with Solidity Naming Conventions.

Category	Severity	Location	Status
Coding Style	 Low	CWW.sol: L: 568 C:29, L: 622, C:11	 Detected

### Description

Solidity defines a naming convention that should be followed. Rule exceptions: Allow constant variable name/symbol/decimals to be lowercase. Allow \_ at the beginning of the mixed\_case match for private variables and unused parameters.

```
deadAddress  
marketingWalletUpdated
```

### Recommendation

Follow the Solidity naming convention.



### Mitigation

### References:

<https://docs.soliditylang.org/en/v0.4.25/style-guide.html#naming-convention>

Writing Clean Code for Solidity: Best Practices for Solidity Development

## cww-07 | State Variables could be Declared Constant.

Category	Severity	Location	Status
Coding Style	 Low	CWW.sol: L: 568 C:14	 Detected

### Description

Constant state variables should be declared constant to save gas.

```
deadAddress
```

### Recommendation

Add the constant attribute to state variables that never changes.


### Mitigation

#### References:

<https://docs.soliditylang.org/en/latest/contracts.html#constant-state-variables>

Writing Clean Code for Solidity: Best Practices for Solidity Development

## cww-08 | Dead Code Elimination.

Category	Severity	Location	Status
Coding Style	<span style="color: green;">●</span> Low	CWW.sol: L: 13 C: 0	 Detected

### Description

Functions that are not used in the contract, and make the code s size bigger.

ABIEncoderV2

### Recommendation

Remove unused functions. dead-code elimination (also known as DCE, dead-code removal, dead-code stripping, or dead-code strip) is a compiler optimization to remove code which does not affect the program results. Removing such code has several benefits: it shrinks program size, an important consideration in some contexts, and it allows the running program to avoid executing irrelevant operations, which reduces its running time. It can also enable further optimizations by simplifying program structure.


### Mitigation

#### References:

Cheatsheetl

Writing Clean Code for Solidity: Best Practices for Solidity Development

## cww-11 | Reentrancy Vulnerability in External Contract Interactions..

Category	Severity	Location	Status
Optimization	<span style="color: orange;">●</span> High	CWW.sol: L: 0 C: 0	 Detected

### Description

The contract contains several functions (swapTokensForEth, addLiquidity, and swapBack) that interact with external contracts (Uniswap V2 Router) and perform Ether transfers to external addresses. These interactions could potentially be exploited by a reentrancy attack if the called contracts perform unexpected actions, such as calling back into the calling contract before the initial execution is complete. This could lead to unintended effects, such as manipulating contract state, draining funds, or disrupting intended logic flow.

### Recommendation



To mitigate the risk of reentrancy attacks, the following steps should be taken: Checks-Effects-Interactions Pattern: Ensure that all state changes occur before any external calls or Ether transfers. This pattern should be strictly followed to prevent attackers from taking advantage of intermediate states. Reentrancy Guard: Implement a reentrancy guard, such as the nonReentrant modifier provided by OpenZeppelin's ReentrancyGuard contract. This modifier should be used on all functions that make external calls or Ether transfers. External Call Review: Review all external calls to ensure they do not inadvertently call untrusted contracts or pass control flow to external entities that could perform a reentrant call. Testing and Auditing: Thoroughly test the contract functions that interact with external contracts to ensure they are not vulnerable to reentrancy. Additionally, a professional audit should be conducted to identify and address any reentrancy vectors. By implementing these remediation steps, the contract can be better protected against reentrancy attacks, ensuring the integrity and security of its operations.

### Mitigation

### References:

## Writing Clean Code for Solidity: Best Practices for Solidity Development

## ! cww-12 | Centralization Risks In The autoBurnLiquidityPairTokens Role or Function.

Category	Severity	Location	Status
Centralization / Privilege	 High	CWW.sol: L: 1066 C:14	 Detected

### Description

In the contract CWW.sol, the role autoBurnLiquidityPairTokens has authority over the following functions:

- function burn(), to burn anyone's account at any amount.

- function burnFrom(), to burn anyone's account at the number in the range of \_allowed .

Any compromise to the autoBurnLiquidityPairTokens account may allow the hacker to take advantage of this authority.

We understand the autoBurnLiquidityPairTokens role could be assigned to the smart contract , however, the

- autoBurnLiquidityPairTokens is a map and more addresses could be added.

### Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation

- and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the

- client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In

- general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized



- mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets.

### Mitigation

### References:

Writing Clean Code for Solidity: Best Practices for Solidity Development

## ■ cww-14 | Unnecessary Use Of SafeMath.

Category	Severity	Location	Status
Logical Issue	 Medium	CWW.sol: L: 261 C: 14	 Detected

### Description

The SafeMath library is used unnecessarily. With Solidity compiler versions 0.8.0 or newer, arithmetic operations

will automatically revert in case of integer overflow or underflow.

library SafeMath {

An implementation of SafeMath library is found.

using SafeMath for uint256;

SafeMath library is used for uint256 type in contract.

### Recommendation

We advise removing the usage of SafeMath library and using the built-in arithmetic operations provided by the

Solidity programming language.



### Mitigation

#### References:

Writing Clean Code for Solidity: Best Practices for Solidity Development



## I cww-18 | Stop Transactions by using Enable Trade.

Category	Severity	Location	Status
Logical Issue	 Critical	CWW.sol: L: 695 C: 14	 Detected

### Description

Enable Trade is present on the following contract and when combined with Exclude from fees it can be considered a whitelist process, this will allow anyone to trade before others and can represent and issue for the holders.

### Recommendation


We recommend the project owner to carefully review this function and avoid problems when performing both actions.

### Mitigation

### References:

Writing Clean Code for Solidity: Best Practices for Solidity Development

## ■ cww-19 | Centralization Privileges of cww.

Category	Severity	Location	Status
Coding Style	<span style="color: orange;">●</span> Medium	CWW.sol: L: 0 C: 0	 Detected

### Description

In a smart contract, the concept of "onlyOwner" functions refers to certain functions that can only be executed by the owner or creator of the contract. These functions are typically designed to perform critical actions or modify sensitive data within the contract. By restricting access to these functions, the contract owner maintains control and ensures the integrity and security of the contract.

Function Name	Parameters	Visibility
Renounce Ownership	renounceOwnership	public
Transfer Ownership	transferOwnership	public
Enable Trading	enableTrading	external
Remove Limits	removeLimits	external
Disable Transfer Delay	disableTransferDelay	external
Update Swap Tokens At Amount	updateSwapTokensAtAmount	external
Update Max Transaction Amount	updateMaxTxnAmount	external
Update Max Wallet Amount	updateMaxWalletAmount	external
Exclude From Max Transaction	excludeFromMaxTransaction	public
Update Swap Enabled	updateSwapEnabled	external
Update Buy Fees	updateBuyFees	external

Function Name	Parameters	Visibility
Update Sell Fees	updateSellFees	external
Exclude From Fees	excludeFromFees	public
Set Automated Market Maker Pair	setAutomatedMarketMakerPair	public
Update Marketing Wallet	updateMarketingWalletInfo	external
Update Development Wallet	updateDevelopmentWalletInfo	external
Set Auto LP Burn Settings	setAutoLPBurnSettings	external
Manual Burn Liquidity Pair Tokens	manualBurnLiquidityPairTokens	external

## Recommendation

Inheriting from Ownable and calling its constructor on yours ensures that the address deploying your contract is registered as the owner. The onlyOwner modifier makes a function revert if not called by the address registered as the owner. It is important that deployer or owner secure the credentials that has owner privilege to ensure the security of the project.



## Mitigation

### References:

[Guide to Ownership and Access Control in Solidity](#)

[Writing Clean Code for Solidity: Best Practices for Solidity Development](#)

## ■ cww-22 | LP Token Burn Configuration and Execution.

Category	Severity	Location	Status
Coding Style	 High	CWW.sol: L: 1048-1088	 Detected

### Description

This smart contract includes functionality related to LP token burning. Here are the key components:

1. Setting Automatic LP Burn Parameters (setAutoLPBurnSettings):
  - The contract owner can configure automatic LP burn settings.
  - Parameters include the burn frequency (minimum every 10 minutes) and the percentage of tokens to burn (between 0% and 10%).
  - Lack of validation for the LP pair address introduces a security risk.
2. Automated LP Token Burning (autoBurnLiquidityPairTokens):
  - Internally executed function.
  - Calculates the amount of tokens to burn from the liquidity pair based on a percentage.
  - Risk lies in accurate calculations and potential liquidity loss.
  - No upper limit on the burn amount.
3. Manual LP Token Burning (manualBurnLiquidityPairTokens):
  - External function restricted to the contract owner.
  - Manually burns tokens from the liquidity pair.
  - Includes a cooldown period to prevent frequent manual burns.
  - Similar risks as automated burning, but with additional owner control.

### Recommendation

Validate\_LP\_Pair\_Address": "Ensure that the LP pair address corresponds to a valid liquidity pool.", "Multi-Signature\_Control": "Implement multi-signature control for owner access to prevent unauthorized changes.", "Maximum\_Burn\_Limit": "Limit the maximum burn amount (e.g., a percentage of total supply).", "Secure\_Execution": "Use well-audited token burn libraries or contracts for secure execution..






### Mitigation

## References:

Writing Clean Code for Solidity: Best Practices for Solidity Development

## FINDINGS

In this document, we present the findings and results of the smart contract security audit. The identified vulnerabilities, weaknesses, and potential risks are outlined, along with recommendations for mitigating these issues. It is crucial for the team to address these findings promptly to enhance the security and trustworthiness of the smart contract code.

Severity	Found	Pending	Resolved
 Critical	1	1	0
 High	2	2	0
 Medium	4	4	0
 Low	5	5	0
 Informational	0	0	0
Total	12	12	0

In a smart contract, a technical finding summary refers to a compilation of identified issues or vulnerabilities discovered during a security audit. These findings can range from coding errors and logical flaws to potential security risks. It is crucial for the project owner to thoroughly review each identified item and take necessary actions to resolve them. By carefully examining the technical finding summary, the project owner can gain insights into the weaknesses or potential threats present in the smart contract. They should prioritize addressing these issues promptly to mitigate any risks associated with the contract's security. Neglecting to address any identified item in the security audit can expose the smart contract to significant risks. Unresolved vulnerabilities can be exploited by malicious actors, potentially leading to financial losses, data breaches, or other detrimental consequences. To ensure the integrity and security of the smart contract, the project owner should engage in a comprehensive review process. This involves understanding the nature and severity of each identified item, consulting with experts if needed, and implementing appropriate fixes or enhancements. Regularly updating and maintaining the smart contract's codebase is also essential to address any emerging security concerns. By diligently reviewing and resolving all identified items in the technical finding summary, the project owner can significantly reduce the risks associated with the smart contract and enhance its overall security posture.

## SOCIAL MEDIA CHECKS | cetwifweed.

Social Media	URL	Result
Website	<a href="https://www.catwifweed.me/">https://www.catwifweed.me/</a>	Pass
Telegram	<a href="https://t.me/cetwifweed">https://t.me/cetwifweed</a>	Pass
Twitter	<a href="https://twitter.com/cetwifweed">https://twitter.com/cetwifweed</a>	Pass
Facebook		N/A
Reddit	N/A	N/A
Instagram	N/A	N/A
CoinGecko	N/A	N/A
Github		N/A
CMC	N/A	N/A
Email	N/A	Contact
Other		Fail

From a security assessment standpoint, inspecting a project's social media presence is essential. It enables the evaluation of the project's reputation, credibility, and trustworthiness within the community. By analyzing the content shared, engagement levels, and the response to any security-related incidents, one can assess the project's commitment to security practices and its ability to handle potential threats.

### Social Media Information Notes:

**Auditor Notes:** Website needs a bit of improvement.

**Project Owner Notes:**

## ASSESSMENT RESULTS | cetwifweed.

### Score Results

Review	Score
Overall Score	17/100
Auditor Score	45/100

Review by Section	Score
Manual Scan Score	0
SWC Scan Score	31
Advance Check Score	-14

Our security assessment or audit score system for the smart contract and project follows a comprehensive evaluation process to ensure the highest level of security. The system assigns a score based on various security parameters and benchmarks, with a passing score set at 80 out of a total attainable score of 100. The assessment process includes a thorough review of the smart contracts codebase, architecture, and design principles. It examines potential vulnerabilities, such as code bugs, logical flaws, and potential attack vectors. The evaluation also considers the adherence to best practices and industry standards for secure coding. Additionally, the system assesses the projects overall security measures, including infrastructure security, data protection, and access controls. It evaluates the implementation of encryption, authentication mechanisms, and secure communication protocols. To achieve a passing score, the smart contract and project must attain a minimum of 80 points out of the total attainable score of 100. This ensures that the system has undergone a rigorous security assessment and meets the required standards for secure operation.



## AUDIT FAILED



## Important Notes for cww

- No Reentrancy Risk: No external calls in critical functions that could lead to reentrancy attacks.■
- Safe Math by Default: Solidity 0.8.x prevents overflows/underflows.■
- No Gas Limit Concerns: Absence of unbounded loops.■
- Explicit Visibility: Functions and state variables have explicit visibility.■
- No Delegatecall: Delegatecall not used, avoiding related risks.■
- Default Values: Some state variables rely on defaults; explicit initialization could improve clarity.■
- No Oracles: Contract does not use oracles, avoiding external manipulation risks.■
- No External Interactions: Contract does not interact with external contracts, reducing attack surface.■
- Proper Use of Modifiers: onlyOwner modifier used correctly.■
- Error Handling: require statements used for validation

and error handling.■

- ERC20 Compliance: Contract adheres to the ERC20 standard.■
- Recommendations:■
- Consider adding events for ownership transfer in the Ownable contract for transparency.■
- Review and test for any potential front-running issues.■
- Ensure off-chain governance and administrative actions are secure and transparent.■
- Conclusion: The contract appears to follow good practices and the ERC20 standard. No immediate security issues detected in the provided code. However, comprehensive testing and potentially a formal verification should be conducted to ensure security, especially for code



## Appendix

### Finding Categories

#### Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

#### Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

#### Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

#### Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

#### Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

#### Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

#### Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different requirements on the input variables than a setter function.

#### Coding Best Practices

ERC 20 Coding Standards are a set of rules that each developer should follow to ensure the code meets a set of criteria and is readable by all the developers.

## Disclaimer

The purpose of this disclaimer is to outline the responsibilities and limitations of the security assessment and smart contract audit conducted by Bladepool/CFG NINJA. By engaging our services, the project owner acknowledges and agrees to the following terms:

1. Limitation of Liability: Bladepool/CFG NINJA shall not be held liable for any damages, losses, or expenses incurred as a result of any contract malfunctions, vulnerabilities, or exploits discovered during the security assessment and smart contract audit. The project owner assumes full responsibility for any consequences arising from the use or implementation of the audited smart contract. 2. No Guarantee of Absolute Security: While Bladepool/CFG NINJA employs industry-standard practices and methodologies to identify potential security risks, it is important to note that no security assessment or smart contract audit can provide an absolute guarantee of security. The project owner acknowledges that there may still be unknown vulnerabilities or risks that are beyond the scope of our assessment. 3. Transfer of Responsibility: By engaging our services, the project owner agrees to assume full responsibility for addressing and mitigating any identified vulnerabilities or risks discovered during the security assessment and smart contract audit. It is the project owner's sole responsibility to ensure the proper implementation of necessary security measures and to address any identified issues promptly. 4. Compliance with Applicable Laws and Regulations: The project owner acknowledges and agrees to comply with all applicable laws, regulations, and industry standards related to the use and implementation of smart contracts. Bladepool/CFG NINJA shall not be held responsible for any non-compliance by the project owner. 5. Third-Party Services: The security assessment and smart contract audit conducted by Bladepool/CFG NINJA may involve the use of third-party tools, services, or technologies. While we exercise due diligence in selecting and utilizing these resources, we cannot be held liable for any issues or damages arising from the use of such third-party services. 6. Confidentiality: Bladepool/CFG NINJA maintains strict confidentiality regarding all information and data obtained during the security assessment and smart contract audit. However, we cannot guarantee the security of data transmitted over the internet or through any other means. 7. Not a Financial Advice: Bladepool/CFG NINJA please note that the information provided in the security assessment or audit should not be considered as financial advice. It is always recommended to consult with a financial professional or do thorough research before making any investment decisions.

By engaging our services, the project owner acknowledges and accepts these terms and releases Bladepool/CFG NINJA from any liability, claims, or damages arising from the security assessment and smart contract audit. It is recommended that the project owner consult legal counsel before entering into any agreement or contract.

