QuillAudits

Audit Report
August, 2024

For

Table of Content

Table of Content

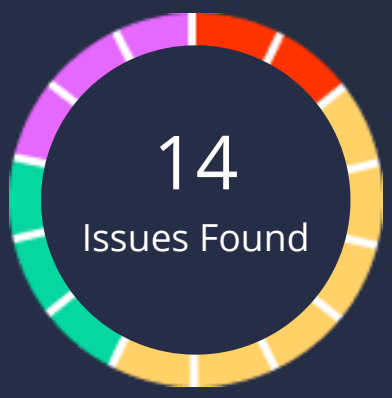
Executive Summary

Project Name	ERUSD
Overview	The ERUSD Protocol, also known simply as ERUSD, introduces a groundbreaking approach to stablecoins with the launch of the ETC Reserve USD (ERUSD). Utilizing Ethereum Classic—one of the pioneering Proof-of-Work blockchains—this protocol enables users to mint ERUSD stablecoins by leveraging Ethereum Classic as approved collateral. Designed to be a resilient financial instrument, ERUSD maintains a soft peg to the US Dollar,ensuring stability and resistance to hyperinflation. This unique combination offers not only economic freedom but also significant global opportunities by mitigating volatility.
Timeline	13th July 2024 - 3rd August 2024
Updated Code Received	4th Spetember 2024
Second Review	9th September 2024 - 10th September 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The Scope of the Audit was to check, security of ERUSD Codebase for vulnerabilities and code quality.
Source Code	https://github.com/VeritasETC/ERSUD-Smart-Contracts/tree/main/contracts **Contracts:** 1]contracts/APY.sol 2]contracts/Actions.sol 3]contracts/ERUSDJoin.sol 4]contracts/ETCJoin.sol 5]contracts/Liquidation.sol 6]contracts/Oracle.sol 7]contracts/TransactionHistory.sol 8]contracts/Vault.sol

Executive Summary

Branch and Commit	Main
Fixed In	6ba1535f7f89c89c9f3e365a954e036edbbdb5c3

Number of Security Issues per Severity



High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	2	1	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	3	2	3

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Manual Review, Foundry, Slither.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Initialization can be performed by anyone.

Path

APY.sol

Function

initialization()

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/APY.sol#L48-L58

Description

The current implementation of the initialization function has no access control. This can be problematic since it is intended to be called only by the factory contract.
The impact is high since initialization is responsible for setting critical functionality and addresses like manager, authentic user, etc.

Recommendation

Add a require check to ensure that it is only called by the factory contract

Status

Resolved

2. Users cannot buy small amounts of ERUSD due to overflow.

Path

Actions.sol

Function

lockAndDraw()

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Actions.sol#L76-L102

Description

The function lockAndDraw is used to provide ETHC in exchange for ERUSD stablecoin with a minimum of 150% collateral ratio.
The problem with this function is its incorrect handling of decimals, resulting in underflow.
Here's how it works:

1. The user provides a tokenAmount along with collateral ratio as input parameter.
2. The call goes to getETHCalculatedAmount and getAmount respectively
    a. https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Actions.sol#L82
3. The getETHCalculatedAmount takes the tokenAmount and _collateralRatio as input parameters and calculate 2 values
    a. _oracleUSDAmount - This is the amount of single eth, i.e. 1e18
    b. requireUSDAmount - The required 150% amount
4. The return value formula is -
    (1 ether * requireUSDAmount) / (_oracleUSDAmount)

Suppose that the user has provided 100 tokens as input. Then in terms of decimals, the calculation becomes:
1e18 * 150 / 1e22

This will round down to zero. Now, as soon as the call goes to the join function, it will revert due to the require check :

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/ETCJoin.sol#L65

Recommendation

Ensure that decimals are correctly handled.

Status

Resolved

Medium Severity Issues

1. Using payable.transfer might be problematic

Description

The usage of transfer so send msg.value is not recommended due to it's strict dependency upon gas i.e. 2300 gas.
If gas costs are subject to change, then smart contracts can't depend on any particular gas costs.

Moreover, the function might fail mid-execution is it requires more than 2300 gas.

Recommendation

Switch to call() instead

Status

Acknowledged

ERUSD Team's Comment

We deliberatly decided to keep the transfer method. Reason being it safe guards against reentrancy attacks. Secondally we donot wanted to write any custom logic for faliure we simple wanted to revert on faliure. For this purpose we kept transfer method.

2. Remaining msg.value is not refunded

Path

Actions.sol

Function

lockAndDraw()

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Actions.sol#L76-L102

Description

Even though lockAndDraw() successfully checks is the provided is sufficient for the calculated amount but it fails to refund the rest of the ETHC.
Suppose that a user provides more ETHC in order to make for volatile gas prices.
In that case, they will not be able to get the remaining msg.value back , will remain in the contract itself.

Recommendation

Ensure that the remaining ETH is transferred back to the user once the execution is completed.

Status

Resolved

3. Malicious users could frontrun as soon as ETCRewardRate is updated.

Path

Oracle.sol

Function

setETHCRate()

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Oracle.sol#L28-L32

Description

The owner has the ability to change ETCRate and can set it to an arbitrary value.
This create problem where everytime the rate is being updated, it can be frontrunned of backrunned based on the rate.

Suppose that ETCRate is increased by owner from 80 to 100. A malicious user watching the mempool frontruns the this transaction and deposit ETH to get stablecoins based on previous rate i.e. 80. Now after the execution of ETCRate transaction, the stablecoins bought by user are worth more and can be convert back to ETCRate to gain profit.

Recommendation

Integrate a price feed that checks the current price everytime it's called.

Status

Acknowledged

ERUSD Team's Comment

Just wanted to highlight the point that front running and back running does not affect the system in the way you mentioned as we do not return the assets to user based on rate
Front running can effect us only in case of liquidation as we liquidate if asset value drops below 137% of loan value. So in this case if some one front run it , only thing he can do is withdraw his assets before liquidation which doesn't effect the platform in negative way.

4. apy amount will underflow is calculate() is called in same day

Path

APY.sol

Function

getCurrentDay()

Description

The getCurrentDay() function returns _currentTime; however, an essential part of the calculation involves counting in days using the formula totalSeconds / 86400. This means that (86400 - 1) is counted as 0 days due to a rounding down error, which may result in incorrect calculations of the day.

Recommendation

To mitigate this issue, count time in terms of block timestamp.

Status

Resolved

5. Import OpenZeppelin contracts directly instead of copying them

Description

The current implementation of ERUSD imports form a Common folder for the dependency.
This approach is not recommended since OpenZeppelin consistently updates its code for improved gas efficiency and possible bugs.

Recommendation

Import openzeppelin contracts as a dependency

Status

Resolved

6. createAPY function doesn’t work properly if apyDetail array length is zero

Path

APYFactory.sol

Function

createAPY();

Description

When authenticUsers call createAPY function to add apycontarct it is underflow due to the index of the apyDetail array going to negative.

Recommendation

update
IAPYMapper(APYMapper).addAPYDetails(address(APYContractClone), _apyPercentage);
Before initialization function call.

Status

Resolved

Low Severity Issues

1. Low test coverage

Description

Unit tests are used to ensure that the code functions as expected by passing in varying user input and setting up various parameters to test the boundaries of the protocol. There are no unit test cases associated with the codebase provided, hereby increasing the probability of bugs being present and reducing quality assurance.

Recommendation

Include unit tests that have > 95% code coverage, including all possible paths for code execution.

Status

Acknowledged

2. Iterating over array.length instead of array.length-1 cause out of bound error

Path

Vault.sol

Function

removeUser()

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Vault.sol#L171-L178

Description

The function removeUser() iterates over loanUsers array to remove the user.
The problem with the implementation is that the for loop is running from 0 to array.length and not array.length - 1
Suppose that there are 5 entries in the array.
When the loop tries to call x = loanUsers.length, it will try to access the entry in the array that is not present and will revert.
The impact is low since the possibility of this happening is rare.

Recommendation

Run the loop to array.length -1

Status

Resolved

3. Lack of zero address check in the constructor might cause problem

Description

Contract initialize/constructor input parameters should always be validated to prevent the creation/initialization of a contract in a wrong/inconsistent state.

Recommendation

Ensure zero address check across all constructors.

Status

Resolved

Informational Issues

1. Unused Mapping

Path

Vault.sol

Function

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/Vault.sol#L33

Description

The mapping is not used anywhere.

Recommendation

It is recommended to remove unused mapping

Status

Resolved

2. Make daysSeconds as an immutable constant instead of state variable

Path

APY.sol

Function

https://github.com/VeritasETC/ERSUD-Smart-Contracts/blob/737f5b92c5652a62049320a5b26dbb51224bbc60/contracts/APY.sol#L33

Description

The daySeconds variable can be set as an immutable constant. By doing so, gas costs of reading this variable will decrease since there will be no reserved storage slot and it will be included in the bytecode.

Recommendation

Mark this as immutable constant

Status

Resolved

3. Contract Manager should not be able to change daysSeconds

Path

APY.sol

Description

The owner of the contract (Manager) is able to change the daysSeconds which should not be the case. This can cause decreased amount of rewards to user than expected if decreased.

Recommendation

Mark it as immutable

Status

Resolved

ERUSD Team's Comment

Contract manager had this power for testing purpose or else the testing of whole contract and process will take days.

Functional Tests Cases

Action.sol

- Should revert if _collateralRatio is greater than minCollateralRatio(): Ensure that the lockAndDraw function reverts when the _collateralRatio parameter provided by the user exceeds the minimum collateral ratio (minCollateralRatio()).
- The lockAndDraw function will revert if taxAmount is greater than msg.value, guaranteeing that the ETH sent is sufficient to cover the stability fee. This validation prevents transactions where the stability fee exceeds the provided ETH.
- Should revert if msg.value is less than or equal to _taxAmount: The lockAndDraw function will revert if the ETH sent (msg.value) is less than or equal to the stability fee (_taxAmount). This check ensures that the user provides sufficient ETH to cover the stability fee, avoiding incomplete transactions.
- Should revert if _ethAmount is not positive: Ensure that the withdrawCollateral function reverts if the amount of ETH (_ethAmount) is zero or less. This check guarantees that there is positive collateral before attempting a withdrawal, thereby avoiding errors when no collateral is available.
- Should revert if there is insufficient collateral in the vault: The withdrawSingleAPYAmount function will revert if the vault does not have enough collateral to fulfill the withdrawal request.

APY.sol

- Should revert if the contract is not live during the deposit function call: The deposit function will revert if the contract is not in the active state, ensuring that deposits are only accepted when the contract is live.
- Should confirm that the calculate function works correctly: The function should accurately perform its intended calculations and return the expected results, validating its correctness and reliability in processing data.

Functional Tests Cases

ERUSD.sol

- Should revert if amount is negative: The function will revert if the provided amount is less than zero, preventing invalid values that could cause errors or unexpected behavior.
- Should save the user's record in the vault and mint the specified amount of tokens to the address: The join function will record the user's details in the vault and mint the provided number of tokens to their address.

ETCJoin.sol

- Should revert if the contract is not live: The function will revert if live is false, indicating that the contract (ETCjoin) must be active for the operation to proceed.
- Should revert if amount is negative: The function will revert if amount is less than zero, ensuring that only non-negative values are processed.

Liquidation.sol

- Should revert if _liqPercent is greater than the minimum collateral ratio: The function will revert if _liqPercent exceeds the value set by IVault(vaultContract).minCollateralRatio(), ensuring compliance with the allowable liquidation percentage.
- Should calculate system health accurately: The getSystemHealth function works correctly by computing the health ratio based on the total collateral and debt. It calculates the system health as the ratio of the USDT value of total collateral to the total debt, ensuring that the system's financial stability is correctly assessed. If there is no collateral, it returns a health value of 0.
- Should return accurate liquidation details: The getLiquidationDetail function correctly calculates and returns the user's collateral and incentive fee, based on current prices and defined rates.
- Should pay the incentive to the master wallet correctly: The liquidateWithSwap function accurately transfers the incentive fee to the master wallet, ensuring proper compensation as per the defined incentive structure.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the ERUSD codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in ERUSD smart contract. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of ERUSD smart contract. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the ERUSD to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com