Blaize

Know the rules.
Create new ones

SMART CONTRACT AUDIT AND SECURITY ANALYSIS REPORT

FOR

TRANCHE COMPOUND PROTOCOL

Table of Contents

This document may contain confidential information about IT systems and the intellectual property of the Customer and information about potential vulnerabilities and methods of their exploitation. The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities are fixed - upon a decision of the Customer.

Abstract

In this report, we consider the security of the **Jibrel** contracts for **Tranche Compound protocol**. Our task is to find and describe security issues in the smart contracts of the platform. This report presents the findings of the security audit of Customer's smart contracts conducted between **March 15th, 2021 - March 23d, 2021.**

Post-audit validation provided on **March, 29th, 2021**.

Disclaimer

The audit does not give any warranties on the security of the code. One audit can not be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, security audits are not investment advice.

Scope

The scope of the audit is the **"tranche-compound-protocol"** project at the main branch with commit 28527d04cf5521b1147f99d156b3f1d2361f128c.

Post-audit scope for validation includes a **"tranche-compound-protocol"** project at the main branch with commit 8fd21f4c973805c47c0fbb8d625957ef891a60f7.

1. ICErc20.sol
2. ICEth.sol
3. IJCompound.sol
4. IJPriceOracle.sol
5. IJTranchesDeployer.sol
6. IJTrancheTokens.sol
7. JCompound.sol
8. JCompoundStorage.sol

9. JTrancheAToken.sol
10. JTrancheBToken.sol
11. JTranchesDeployer.sol
12. TransferETHHelper.sol

We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered (the full list includes them but is not limited to):

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/local variables;
- Compile version not fixed.

Procedure

In our report we checked the contract with the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets best practices in efficient use of gas, code readability;

We perform our audit according to the following procedure:

1. Automated analysis:
    - Scanning contract by several public available automated analysis tools such as Mythril, Solhint, Slither and Smartdec;
    - Manual verification of all the issues found by tools.

2. Manual audit:
    - Manual analysis of smart contracts for security vulnerabilities;
    - Checking smart contract logic and comparing it with one described in the documentation.

Executive summary

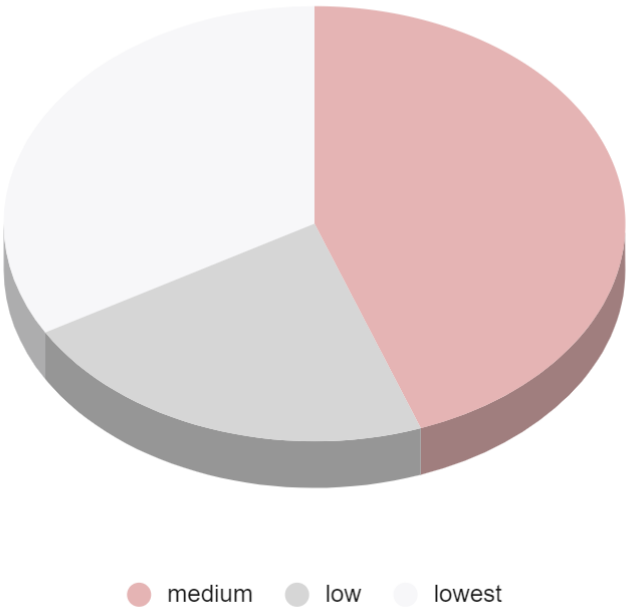
According to the assessment, the Customer's smart contracts required improvements; some functionality works semi-auto and do not follow best practices. We described all the issues and added recommendations for the Customer. Most of the issues found refer to code quality, missing checks and extra code. Though, the Customer's team has provided **all** necessary improvements according to the Auditor's recommendations.

For the overall security of the smart-contracts system can be evaluated as having good security and has **99** out of **100**.

Findings:

	Found	**Fixed**
Critical	**0**	**0**
High	**0**	**0**
Medium	**4**	**4**
Low	**2**	**2**
Lowest	**3**	**3**

The graph of vulnerabilities distribution:



Severity Definition

Critical	A system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Needs immediate improvements and further checking.
High	A system contains a couple of serious issues, which lead to unreliable work of the system and might cause a huge information or financial leak. Needs immediate improvements and further checking.
Medium	A system contains issues which may lead to medium financial loss or users' private information leak. Needs immediate improvements and further checking.
Low	A system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Needs improvements.
Lowest	A system does not contain any issue critical to the secure work of the system, yet is relevant for best

	software defensive practices implementations.

AS-IS overview

JCompound contract overview

JCompound contract inherits from the OwnableUpgradeSafe, JCompoundStorage and IJCompound contracts.

From Initializable

- isConstructor() (private)
- From ContextUpgradeSafe
  - __Context_init() (internal)
  - __Context_init_unchained() (internal)
  - _msgData() (internal)
  - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- Native functions
  - initialize(address,address,address) (public) - sets price oracle address, fees collector address, tranches deployer address
  - fallback() (external) - this is needed to receive ETH when calling redeemCEth function
  - receive() (external) - this is needed to receive ETH when calling redeemCEth function
  - setBlocksPerYear(uint256) (external) - sets how many blocks will be produced per year on the blockchain
  - setCEtherContract(address) (external) - sets relationship between ethers and the corresponding Compound cETH contract
  - setCTokenContract(address, address) (external) - sets relationship between a token and the corresponding Compound cToken contract
  - isCTokenAllowed(address) -> bool (public) - checks if a cToken is allowed or not
  - getCompoundPercentagePerTranche(uint256) -> uint256 (external) - gets percentage from compound

- setDecimals(uint256, uint8, uint8) (external) - checks if a cToken is allowed or not
- setTrancheRedemptionPercentage(uint256, uint16) (external) - sets tranche redemption percentage
- setTrancheAFixedPercentage(uint256, uint256) (external) - sets tranche redemption percentage
- setRedemptionTimeout(uint32) (external) - sets redemption timeout
- addTrancheToProtocol(address,string,string,string,string,uint256,uint8,uint8) (external) - adds tranche in the protocol
- sendErc20ToCompound(address,uint256) -> uint256 (internal) - sends an amount of tokens to corresponding compound contract (it takes tokens from this contract). Only allowed token should be sent
- redeemCErc20Tokens(address,uint256,bool) -> uint256 (internal) - redeems an amount of cTokens to have back original tokens (tokens remain in this contract). Only allowed token should be sent
- redeemCEth(uint256, bool) -> uint256 (internal) - redeems cETH from compound contract (ethers remain in this contract)
- getCEthExchangeRate() -> uint256 (public) - gets cETH exchange rate from compound contract
- getCTokenExchangeRate(address) -> uint256 (public) - gets cToken exchange rate from compound contract
- getMantissa(uint256) -> uint256 (public) - gets tranche mantissa
- getCompoundPrice(uint256) -> uint256 (public) - gets compound price for a single tranche
- setTrancheAExchangeRate(uint256) ->uint256 (public) - sets Tranche A exchange rate
- getTrancheAExchangeRate(uint256) -> uint256 (public) - gets Tranche A exchange rate
- getTrancheACurrentRPB(uint256) -> uint256 (public) - gets RPB for a given percentage (expressed in 1e18)
- calcRPBFromPercentage(uint256) -> uint256 (public) - gets Tranche A exchange rate
- getTrAValue(uint256) -> uint256 (public) - gets Tranche A value
- getTrBValue(uint256) -> uint256 (external) - gets Tranche B value
- getTotalValue(uint256) -> uint256 (public) - gets Tranche total value

- getTrancheBExchangeRate(uint256, uint256) -> uint256 (public) - gets Tranche B exchange rate
- buyTrancheAToken(uint256, uint256) (external) - buys Tranche A Tokens
- redeemTrancheAToken(uint256, uint256) (external) - redeems Tranche A Tokens
- buyTrancheBToken(uint256, uint256) (external) - buys Tranche B Tokens
- redeemTrancheBToken(uint256, uint256) external - redeems Tranche B Tokens
- redeemCTokenAmount(uint256, uint256) (external) - redeems every cToken amount and send values to fees collector
- getTokenBalance(address) -> uint256 (public) - gets every token balance in this contract
- getEthBalance() -> uint256 (public) - gets eth balance on this contract
- transferTokenToOwner(address, uint256) (external) - transfers tokens in this contract to owner address
- withdrawEthToOwner(uint256) (external) - transfers ethers in this contract to owner address

JCompoundStorage contract overview

Inherits from the OwnableUpgradeSafe.

From Initializable

- - isConstructor() (private)

- From ContextUpgradeSafe
  - - __Context_init() (internal)
  - - __Context_init_unchained() (internal)
  - - _msgData() (internal)
  - - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)

JTrancheAToken contract overview

Inherits from the OwnableUpgradeSafe, ERC20UpgradeSafe, AccessControlUpgradeSafe, IJTrancheTokens.

- From Initializable
  - isConstructor() (private)
- From ContextUpgradeSafe
  - __Context_init() (internal)
  - __Context_init_unchained() (internal)
  - _msgData() (internal)
  - _msgSender() (internal)
- From OwnableUpgradeSafe
  - __Ownable_init() (internal)
  - __Ownable_init_unchained() (internal)
  - owner() (public)
  - renounceOwnership() (public)
  - transferOwnership(address) (public)
- From AccessControlUpgradeSafe
  - __AccessControl_init() (internal)
  - __AccessControl_init_unchained() (internal)
  - hasRole(bytes32,address) (public)
  - getRoleMemberCount(bytes32) (public)
  - getRoleMember(bytes32,uint256) (public)
  - getRoleAdmin(bytes32) (public)
  - grantRole(bytes32,address) (public)
  - revokeRole(bytes32,address) (public)
  - renounceRole(bytes32,address) (public)
  - _setupRole(bytes32,address) (internal)
  - _setRoleAdmin(bytes32,bytes32) (internal)
  - _grantRole(bytes32,address) (private)
  - _revokeRole(bytes32,address) (private)
- From ERC20UpgradeSafe
  - __ERC20_init(string,string) (internal)
  - __ERC20_init_unchained(string,string) (internal)
  - name() (public)
  - symbol() (public)
  - decimals() (public)
  - totalSupply() (public)
  - balanceOf(address) (public)
  - transfer(address,uint256) (public)
  - allowance(address,address) (public)
  - approve(address,uint256) (public)
  - transferFrom(address,address,uint256) (public)
  - increaseAllowance(address,uint256) (public)
  - decreaseAllowance(address,uint256) (public)

    - _transfer(address,address,uint256) (internal)
    - _mint(address,uint256) (internal)
    - _burn(address,uint256) (internal)
    - _approve(address,address,uint256) (internal)
    - _setupDecimals(uint8) (internal)
    - _beforeTokenTransfer(address,address,uint256) (internal)
- Native functions
    - initialize(string, string) (public) - initializes OwnableUpgradeSafe and ERC20UpgradeSafe and sets up the minter role
    - setJCompoundMinter(address) (external) - grants the minter role to the specified account
    - mint(address, uint256) (external) - mints tokens to an account
    - burn(uint256) (external) - burns an amount of the token of a given account

JTrancheBToken contract overview

Inherits from the OwnableUpgradeSafe, ERC20UpgradeSafe, AccessControlUpgradeSafe, IJTrancheTokens.

- From Initializable
    - isConstructor() (private)
- From ContextUpgradeSafe
    - __Context_init() (internal)
    - __Context_init_unchained() (internal)
    - _msgData() (internal)
    - _msgSender() (internal)
- From OwnableUpgradeSafe
    - __Ownable_init() (internal)
    - __Ownable_init_unchained() (internal)
    - owner() (public)
    - renounceOwnership() (public)
    - transferOwnership(address) (public)
- From AccessControlUpgradeSafe
    - __AccessControl_init() (internal)
    - __AccessControl_init_unchained() (internal)
    - hasRole(bytes32,address) (public)
    - getRoleMemberCount(bytes32) (public)
    - getRoleMember(bytes32,uint256) (public)
    - getRoleAdmin(bytes32) (public)
    - grantRole(bytes32,address) (public)
    - revokeRole(bytes32,address) (public)

    - renounceRole(bytes32,address) (public)
    - _setupRole(bytes32,address) (internal)
    - _setRoleAdmin(bytes32,bytes32) (internal)
    - _grantRole(bytes32,address) (private)
    - _revokeRole(bytes32,address) (private)
- From ERC20UpgradeSafe
    - __ERC20_init(string,string) (internal)
    - __ERC20_init_unchained(string,string) (internal)
    - name() (public)
    - symbol() (public)
    - decimals() (public)
    - totalSupply() (public)
    - balanceOf(address) (public)
    - transfer(address,uint256) (public)
    - allowance(address,address) (public)
    - approve(address,uint256) (public)
    - transferFrom(address,address,uint256) (public)
    - increaseAllowance(address,uint256) (public)
    - decreaseAllowance(address,uint256) (public)
    - _transfer(address,address,uint256) (internal)
    - _mint(address,uint256) (internal)
    - _burn(address,uint256) (internal)
    - _approve(address,address,uint256) (internal)
    - _setupDecimals(uint8) (internal)
    - _beforeTokenTransfer(address,address,uint256) (internal)
- Native functions
    - initialize(string, string) (public) - initializes OwnableUpgradeSafe and ERC20UpgradeSafe and sets up the minter role
    - setJCompoundMinter(address) (external) - grants the minter role to the specified account
    - mint(address, uint256) (external) - mints tokens to an account
    - burn(uint256) (external) - burns an amount of the token of a given account

JTranchesDeployer contract overview

Inherits from the OwnableUpgradeSafe and IJTranchesDeployer.

- From Initializable
    - isConstructor() (private)
- From ContextUpgradeSafe
    - __Context_init() (internal)

    - __Context_init_unchained() (internal)
    - _msgData() (internal)
    - _msgSender() (internal)
- From OwnableUpgradeSafe
    - __Ownable_init() (internal)
    - __Ownable_init_unchained() (internal)
    - owner() (public)
    - renounceOwnership() (public)
    - transferOwnership(address) (public)
- Native functions
    - initialize() (public)
    - setJCompoundAddress(address) (public) - sets `JCompoundAddress`
    - deployNewTrancheATokens(string, string, address) -> address (public) - deploys the JTrancheAToken
    - deployNewTrancheBTokens(string, string, address) -> address (public) - deploys the JTrancheBToken

TransferETHHelper contract overview

- Native functions
    - safeTransferETH(address, uint256) (internal) - saves transfer eth helper

Audit overview

Critical

No critical issues detected.

High

No critical issues detected.

Medium

1. **JCompound.sol**. Unused ownership.

   Evidence: Contract is marked as **Ownable** though there is no application of that functionality.

Recommendation: Confirm that owner should not be checked in *onlyAdmins()* modifier and remove inheritance from **Ownable** contract.

Resolution: Fixed by the Client. Added environment change functionality.

2. **JCompound.sol**. Missing checks.

Evidence: Functions *setDecimals()* and *addTrancheToProtocol()* have no any checks regarding the correctness of decimals set in the contract. The decimals number influences several functions: *getTrBValue()*, *getTrBValue()*, *getTotalValue()*, *getMantissa()* (in the places of calls). There can be an arithmetic error with **overflow** in exponencial operation (**10 ** decimals**).

In general there is a low probability of mistake, but due to the fact, that these functionality affects several places in the contract, that it is overflow possibility, that there can be different number of decimals (starting with standard USDT with 6 decimals, which can cause errors in calculation) we recommend to check the math and add security checks.

Also, all automatic tools we have used marked this issue as high. It is added to the **medium** section only because of low probability of error setup in admin's method.

So we recommend adding checks, that *trancheParameters[_trancheNum].underlyingDecimals* can never be set higher than **18**, and that after decimals setting *getMantissa*() method will never return the number greater than **18**.

Recommendation: Add security checks for the correct decimals setup.

Resolution: Client has added appropriate security checks and mantissa calculation fixes.

3. **JCompound.sol**. Missing error message.

   Evidence: locked() modifier has no error message in the required statement. In general it is a low risk issue. Though it is a modifier that acts as Reentrancy guard, so it has very high meaning for the system and should reflect its own behavior with appropriate messages. That's why the issue is marked as **medium**.

   Recommendation: Add error message.

   Resolution: Fixed by the Client.

4. **JCompound.sol**. Unused storage variable.

   Evidence: setCEtherContract() sets value of cEth contract twice - to *cEtherContract* and *cEthToken*. Though *cEtherContract* is not used is the contract set.

   Recommendation: Remove unused variables.

   Resolution: Fixed by the Client.

Low

1. **JTrancheAToken.sol, JTrancheBToken.sol**. Additional checks.

   Evidence: mint() and *burn()* methods do not contain checks for zero amount. It is a low probability of calling these methods with zero amount, though such checks can prevent standard accidental transactions with 0 amount, which will have no effect but cost gas.

   Recommendation: Add checks for 0 amount.

   Resolution: Fixed by the Client.

2. **JCompound.sol**. Incorrect naming and documentation.

   Evidence: transferTokenToOwner() and *withdrawEthToOwner()* names and docstrings state that these methods transfer funds to the

contract's owner. Though they transfer funds to the fee collector address.

Recommendation: Correct naming and documentation and check that the receiver is correct.

Resolution: Fixed by the Client.

Lowest

Informational statements

1. **JTranchesDeployer.sol**. Extra import.

   Evidence: ICompound.sol and *IERC20.sol* are imported but never used.

   Recommendation: Remove unnecessary imports.

   Resolution: Fixed by the Client.

2. **JTrancheAToken.sol, JTrancheBToken.sol**. Misleading documentation.

   Evidence: mint() and *burn()* methods contain misleading information about *pointsCorrection* update.

   Recommendation: Remove unnecessary imports.

   Resolution: Fixed by the Client.

3. **JCompound.sol**. Magic number.

   Evidence: redeemTrancheAToken() and *redeemTrancheBToken()* methods contain "magic" number for precision - 10000. It will be more obvious and safer for further development to move it into the constant.

   Recommendation: Move number to the constant.

   Resolution: Fixed by the Client.

Unit Test Coverage

All present tests can be successfully run. Nevertheless, the non-standard approach for test writing does not allow to check the test coverage in an automatic way, so manual review for tests coverage was applied. The Auditor's team has considered that the project has sufficient test coverage.

Conclusion

According to the audit the contract was manually reviewed and analyzed with static analysis tools. The Audit team has found some **medium** and low issues during the analysis. All issues should be fixed by the Customer's team following Auditor's recommendations. Most of the issues found refer to code quality, missing checks and extra code. Though, all issues were fixed by the Customer's team following Auditor's recommendations. Due to the foundings and fixes, the contracts system can be marked as **secure**.

The overall security of the smart-contracts system can be evaluated as **99** out of **100**.

Audit report contains all necessary information related to it as well as recommendations for their elimination.