Cheelee Security Analysis

by Pessimistic

This report is public

December 29, 2022

Abstract

In this report, we consider the security of smart contracts of Cheelee project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot 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, a security audit is not investment advice.

Summary

In this report, we considered the security of Cheelee smart contracts. We described the audit process in the section below.

The initial audit showed several critical issues: Bug in vesting release, Incorrect reward calculation, Possibility of deleting vesting info. The initial audit also revealed several issues of medium severity: No guarantees for vested tokens, No tests for crucial scenarios, as well as two Reentrancy issues (M03, M04). Moreover, several low-severity issues were found.

The quality of the code is mediocre. The developers follow the best practices, however, the initial codebase contained various serious issues, and the code was not ready for the deployment.

In the codebase update, all of the critical and most of the medium and low severity issues were fixed. However, tests are still not covering crucial scenarios (M02) and developers introduced two new issues of medium severity: M06 and M07.

In the second codebase update, issues M06 and M07 were fixed. Moreover, the developers also fixed the remaining low severity issues. However, this update contained a new bug, which was fixed later.

In the third codebase update, the developers made several contracts upgradeable. We checked the correctness of the new code and added one issue of low severity.

General recommendations

We recommend writing additional tests.

Project overview

Project description

For the audit, we were provided with Cheelee project on a public GitHub repository, commit ed94d2e1bc36634fd725fabd8e3e283617e6d8f9.

The scope of the audit includes everything.

The up-to-date documentation for the project includes `Tech-samrt-contracts-v0.5.pdf`, sha1sum adc3623583d3674ca8b4ed5143bf188e75d242b8. The initial version of the documentation contained a discrepancy with the code (see M05 issue), but it was fixed shortly after the audit.

All 19 tests pass successfully. The code coverage is 82.5%.

The total LOC of audited sources is 867.

Codebase update #1

After the audit, we were provided with commit edf8c77d00f2e1fc004fe5723dc0c7e2a0b3497c. In the update the developers fixed most of the issues, implemented new tests, as well as added comments to the code. However, the developers introduced two new issues of medium severity: M06 and M07. In addition to this, tests are still not covering crucial scenarios.

All 22 tests pass successfully. The code coverage is 94.63%.

Codebase update #2

After the code recheck, the developers provided commit 049fe1b1b6e3af0cdfa5854e6cdb662bdd3cd9b5. In the update, the developers fixed issues introduced in the previous codebase update. Moreover, they fixed several low severity issues found during the initial audit. However, the developers introduced a new bug, which was fixed later in commit 7959e95da68330ba57e027eb86a6d19227d6044c. The developers changed tests for **MultiVesting** contract to match the new functionality, however we still suggest adding more tests for other parts of the project.

All 22 tests pass successfully. The code coverage is 94.98%.

Codebase update #3

For the third update, the developers provided commit 9ee167c7ec024344df9854ec29187131ed0ebf5d. In the update, the developers added upgradeability functionality and covered it with tests.

All 25 tests pass successfully. The code coverage is 95.67%.

Audit process

We started the audit on November 15 and finished on November 25, 2022.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among other, we verified the following properties of the contracts:

- We checked the correctness of the formulas used in staking and vesting contracts (see critical issues).
- We checked that tokens from the treasury could not be stolen.
- We checked whether the project complies with the documentation (see Discrepancy with the documentation issue).

We scanned the project with the static analyzer Slither with our own set of rules and then manually verified all the occurrences found by the tool.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

After the initial audit, we received a new commit with the updated codebase.

We checked if the issues from the initial audit were fixed. We also reviewed new tests and whether they cover scenarios from critical issues found during the initial audit. We also checked whether the final version of the documentation complies with the code.

After that, we updated the report with the new found issues and notes on previous ones.

After the recheck, we made a call with the developers and discussed new found issues and possible solutions for them. Afterwards we received a new version of the code. We checked the fixes for the remaining issues and whether new functionality introduced new ones. We notified developers about a new bug and checked how it was fixed later.

After the second recheck, the developers contacted us to ensure that Openzeppelin upgradeable library is used correctly. We did not find any major issues. We recommended preventing implementation contract initialization as described in L14 issue. The issue was fixed later.

We combined our work result in the current version of the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Bug in vesting token release (fixed)

In **MultiVesting.sol** there is a bug in `release` function, that allows draining all funds from the contract. Line 80 calculates releasable amount and uses `msg.sender` as a `_beneficiary` argument for `_releasable` function call. However, this releasable value is not added to `msg.sender's` released amount, but to an arbitrary address' at line 87. As a result, a malicious user is able to provide `_beneficiaryAddress` different from the `msg.sender` and their released value would not be updated.

The issue has been fixed and is not present in the latest version of the code.

C02. Incorrect reward calculation (fixed)

In **Staking** contract, line 176 should update the amount of already claimed tokens. This value is used at line 168 in `earned` function: the new reward is equal to the total unlocked reward amount minus the amount of already claimed tokens. In reality, line 176 uses `=` operator instead of `+=`. That means `alreadyCollected` variable stores the amount of tokens unlocked during the previous call to `_collect` function and not the amount of claimed tokens by the user.

This bug allows to drain all funds from the contract. Imagine malicious user sends 3 transactions with the call to `collect` function in the same block:

1. The first transaction works as expected, and the user receives all unlocked reward. Since `status[_option][_addr].alreadyCollected` equals to zero, `_earned` value and line 168 equals to all unlocked reward and user receives in. Note, that `status[_option][msg.sender].alreadyCollected` value is set to `_earned` value, i.e. all available reward.
2. The second transaction also works as expected since `status[_option][msg.sender].alreadyCollected` value equals to `_amount` at line 168 and `_earned` amount equals to zero. However, line 176 sets `alreadyCollected` variable to zero.
3. That is why the third transaction will work in the same way as the first one: since `alreadyCollected` amount is zero, the user will receive all unlocked rewards.

The issue has been fixed and is not present in the latest version of the code.

C03. Possibility of deleting vesting info (fixed)

In **MultiVesting.sol**, anyone is able to call `updateBeneficiary` function and provide an arbitrary `_newBeneficiary` argument. As a result, information about the vesting of `_newBeneficiary` will be overridden by the information of `_oldBeneficiary`. Not only this allows to reduce reward for the `_newBeneficiary`, but also to delete all information about it in case when `beneficiary[_oldBeneficiary]` is empty.

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. No guarantees for vested tokens (fixed)

In the code there is no guarantees that the balance of **MultiVesting** contract is sufficient for the vesting. Since total vested amount can be increased at any time with the `vest` function, we recommend ensuring that the contract contains enough tokens.

The issue has been fixed and is not present in the latest version of the code.

M02. No tests for crucial scenarios

The code coverage is 82.5%. Usually, we consider that as a good coverage. However, despite the high coverage, the code contains some crucial bugs in default scenarios (see critical issues). That is why we recommend reviewing and rewriting tests in a better way.

In codebase updates the developers increased code coverage to 94.98%. Still, we recommend covering more scenarios.

M03. Reentrancy (fixed)

In **NFTSale** contract, it is possible to bypass the check of redeem supply at line 109 with the reentrancy attack. Line 124 contains `receiveNFT` call that invokes `onERC721Received` hook. Since this is an external call, this hook allows to reenter the contract prior to redeem supply update at line 125. We recommend updating the `redeemed` variable prior to external calls in order to mitigate possible reentrancy attack and following the CEI pattern in `redeem` function of **NFTSale** contract.

The issue has been fixed and is not present in the latest version of the code.

M04. Reentrancy (fixed)

In **NFTSale** contract, it is possible to bypass the check of purchase supply at line 134 with the reentrancy attack. Line 150 contains `receiveNFT` call that invokes `onERC721Received` hook. Since this is an external call, this hook allows to reenter the contract prior to redeem supply update at line 151. We recommend updating the `purchased` variable prior to external calls in order to mitigate possible reentrancy attack and following the CEI pattern in `purchase` function of **NFTSale** contract.

The issue has been fixed and is not present in the latest version of the code.

M05. Discrepancy with the documentation (fixed)

The documentation states that the owner of `CHEEL` and `LEE` tokens should be able to burn tokens from any address. However, in the code, the owner is able to burn tokens only from his address. We contacted the developers, and they confirmed that the correct behavior was implemented in the code. Still, we recommend fixing the documentation.

The developers updated the documentation.

M06. Frontrunning the vesting schedule setup (fixed)

There are possible frontrun attacks on the `vest` function in **MultiVesting** contract with the use of `updateBeneficiary` function.

Malicious user is able to prevent changes to their vesting schedule by moving their vesting to a new address prior to the call of `vest` function by the owner. As a result, the transaction with the `vest` function will fail as it tries to change the vesting schedule of an old account.

Moreover, a malicious user is able to prevent one other user from participating in vesting. They can fill other user's vesting information with their own by calling `updateBeneficiary` function. As a result, the attempt to vest tokens to a new user will fail since they already have some vesting information. Note that this issue can be resolved by either affected user or owner moving that vesting to some other address.

The issue has been fixed and is not present in the latest version of the code.

M07. Code logic (fixed)

In **MultiVesting** contract, `vest` function checks if the contract holds `sumVesting + _amount` tokens and the `_amount` is added to the `sumVesting`. However, `sumVesting` is not subtracted when tokens are released in `release` function. This means `sumVesting` does not account for released tokens, and the contract will require to hold tokens more than needed in case when owner adds new vesting after somebody already claimed a reward.

The issue has been fixed and is not present in the latest version of the code.

M08. Bug in vesting cliff check (fixed)

There is a bug in **MultiVesting** contract. Lines 102-107 contain a check for the cliff period. According to revert message, the statement ensures that new cliff ends no later than the previous one in case of vesting schedule change. However, this statement has incorrect comparison operator. Moreover, this check can be only reached in case when `beneficiary[_beneficiaryAddress].amount` is zero, i.e. only during vesting schedule initialization. Hence, this check is redundant in the current version of the code.

This issue has been fixed with commit 7959e95da68330ba57e027eb86a6d19227d6044c.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Bad event naming (fixed)

In **NFTSale** contract, the `Pause` event is used for pausing redeems, and the `Redeem` event is emitted when purchases are paused. These event names might be misleading. Consider renaming them.

The issue has been fixed and is not present in the latest version of the code.

L02. Code logic (fixed)

In **MultiVesting.sol**, `vest` function overrides the user's vesting `start`, `duration`, and `cliff` parameters, even if the user has already participated in vesting process. As a result, there might be the following situations:

- If the user cliff period from the previous vesting has ended, they might need to wait for a new one in case they receive more tokens via `vest` function. This happens due to the check at line 69.
- Since `vest` function overwrites both vesting `start` and `duration` parameters, this can reduce the speed of token vesting despite the higher total amount. As a result, the user might stop receiving tokens for some time in case when the previous vesting speed was higher, and the user should have received more tokens at some time according to previous vesting parameters.

Moreover, these issues might cause unexpected reverts due to line 112 in case when the user claimed something prior to changing vesting schedule.

The owner still has an ability to change the vesting schedule. However, the developers introduced several restrictions: vesting amount cannot be changed and new vesting cliff should end no later than the previous one. Moreover, they fixed the unexpected reverts issue by adding an additional check for that case. To sum up, we consider this issue as fixed.

L03. Code quality (commented)

In the project, there are hardcoded variables `GNOSIS`, that are marked as `constant`. We recommend avoiding using hardcoded addresses and marking them as `immutable` in order to make the testing process easier.

> *The developers decided to use hardcoded addresses since gnosis wallets are already deployed. Moreover, this approach negates the risk of changing these addresses during the deployment.*

L04. Code style (fixed)

The common way to add token URI to the NFT is to override `_baseURI` method from **ERC721.sol**. Instead, the code overrides `tokenURI` method in **NFT** contract. Moreover, OpenZeppelin's version of `tokenURI` function checks that the token with the provided `tokenId` exists. The version in **NFT.sol** is able to return a valid URI for the nonexistent token, which may lead to potential problems on offchain integrations.

The issue has been fixed and is not present in the latest version of the code.

L05. Constants (fixed)

Variables `Name`, `EIP712_VERSION`, `NFT_PASS_TYPEHASH`, and `PASS_TYPEHASH` are available at the compile time and do not need to be changed later. Consider making them constant in **Treasury** contract.

The issue has been fixed and is not present in the latest version of the code.

L06. Excessive inheritance list (fixed)

`ERC721Enumerable` already inherits from the `ERC721`. We recommend removing `ERC721` from the inheritance list. Besides, it will allow removing overridden boilerplate functions: `_beforeTokenTransfer`, `supportsInterface` at line 10 in **NFT** contract.

The issue has been fixed and is not present in the latest version of the code.

L07. Mark variable as immutable (fixed)

Variable `nftContract` is set during deployment and never changes again. Consider marking it as `immutable` in order to reduce gas consumption at line 34 in **NFTSale** contract.

The issue has been fixed and is not present in the latest version of the code.

L08. Mark arguments as indexed (fixed)

We recommend marking the following arguments as indexed in order to increase user experience:

- `tokenId` and `receiver` arguments in `ReceiveNFT` event in **NFT.sol**
- `beneficiary` and `oldBeneficiary` arguments in `UpdateBeneficiary` event in **MultiVesting.sol**
- `beneficiary` argument in `Vested` event in **MultiVesting.sol**

The issues have been fixed and are not present in the latest version of the code.

L09. Missing check (fixed)

In **Staking.sol** in `getRegisteredUsersSample` function we recommend checking that `_from` parameter is lower than `_to` and `_to` parameter is lower than `registeredUsers` length.

The issue has been fixed and is not present in the latest version of the code.

L10. Missing check (fixed)

In **Staking.sol** in `setOption` and `addOption` functions, we recommend checking that `_minValue` is not greater than `_maxValue`.

The issue has been fixed and is not present in the latest version of the code.

L11. Missing check (fixed)

In **Staking.sol** in `deposit` function we recommend checking that provided option number is valid. If a user provides an incorrect option number with the `amount` equal to zero, the function will not revert, which might be confusing.

The issue has been fixed and is not present in the latest version of the code.

L12. No NatSpec (fixed)

The codebase of the project does not have any comments. There are no descriptions for contracts and functions. We recommend covering the code with `NatSpec` comments, as it helps to avoid errors and accelerates the development process in the project.

The issue has been fixed and is not present in the latest version of the code.

L13. Redundant code (fixed)

According to `receiveNFT` function in **NFT** contract, the caller of `receiveNFT` should own NFT token if that token exists (line 39). That is why in **Treasury.sol** , lines 100-101 are redundant since treasury should already own tokens in scenarios where `approval` is applicable.

The issue has been fixed and is not present in the latest version of the code.

L14. Initialization of the implementation contract (fixed)

The common pattern for upgradeable contracts is to move all the contract initialization into `initialize` function. This allows to initialize proxy storage. However, there is an attack vector that exploits `initialize` function by making a call directly to the logic contract. The current implementation is safe from this type of attack. Nevertheless, we recommend preventing logic contract initialization in order to make future versions of the code more secure.

The issue has been fixed with commit eb85b9b1e0b22e7b8851aad16fa4b9136863d280.

Notes

N01. Overpowered role

The system relies heavily on the following roles:

- In **NFT** contract:
  - Owner can:
    - mint new tokens
    - update the token URI
    - update the NFTSale and Treasury addresses
  - NFTSale, Treasury can mint new tokens.
- In **NFTSale** contract:
  - Owner can:
    - set a signer
    - update redeem and purchase supplies
    - pause redeems and purchases
  - Signer can sign messages that allow a user to purchase or redeem an NFT token.
- In **Staking** contract, the owner can add a new staking option, modify or pause existing staking options.
- In **Treasury** contract:
  - Owner can:
    - set a signer
    - set withdrawal limits on tokens
    - update the proxy contract
  - Signer can sign messages that allow to withdraw tokens from the treasury.

- In **MultiVesting** contract:
    - Owner can:
        - set a seller
        - withdraw all funds on emergency situations
        - disable emergency withdrawals
    - Seller can vest new tokens.
- In **CHEEL** contract, the owner can mint new tokens.
- In **LEE** contract, the owner can mint new tokens.

Note that the project developers transfer ownership to the Gnosis Safe wallet.

> *Comment from the developers:*
>
> *The team of Cheelee has been working on the resolution of centralization risk to ensure the transparency and authenticity of our business.*
>
> *Gnosis Safe was chosen as it is the most trusted platform to manage digital assets with their multi-signature format for crypto projects and teams.*
>
> *The purpose of having signatures to confirm every transaction is to lower the risk of having unauthorized access to company crypto assets and all Cheelee smart contracts. There are a total of 3 signatures from 5 required to authorize transactions in order to enhance the security in Cheelee as well as keeping the smart contracts decentralized.*

N02. **Code logic**

In **Treasury.sol** users must provide a signature in order to withdraw tokens. The signature includes the nonce field. This field prevents a malicious user from using the same signature twice. However, usually, nonces allow to invalidate old signatures. For example, in Ethereum transactions with nonces less than 15 cannot be mined if transaction with nonce 14 has been mined. Note that in **Treasury.sol** there is a mechanism for invalidating old signatures: each signature has ttl. The problem with that approach is that developers should not give the user another signature for withdrawing NFT until the previous one is invalidated due to ttl.

> *Comment from the developers: Developers do not allow to have another signature for the same amount of tokens or an NFT during the TTL of a previous signature. However it is out of scope of the contracts and provided by the backend infrastructure of the project.*

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Pavel Kondratenkov, Security Engineer
Yhtyyar Sahatov, Junior Security Engineer
Irina Vikhareva, Project Manager

December 29, 2022