Quail Contracts

Security Review

Review by:
Blockdev, Security Researcher

April 2, 2024

Contents

1 Introduction

1.1 Disclaimer

A security review a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.2 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.2.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Quail finance facilitates collateral-free loans to community members based on social trust.

From Mar 22nd to Apr 24th the security researchers conducted a review of QuailContracts on commit hash e435d219. The team identified a total of **27** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 9
- Medium Risk: 4
- Low Risk: 5
- Gas Optimizations: 2
- Informational: 6

3 Findings

3.1 Critical risk

3.1.1 `potCreator` has a way to always win a round

Severity: Critical risk

Context: QuailFinance.sol#L97-L101, QuailFinance.sol#L145-L146

Description: Once `entropy.request()` is called and `sequenceNumber` is known, `potCreator` knows the random number that will be generated through `entropy.reveal()`. Thus, `potCreator` can predict `winnerIndex` before calling `rotateLiquidity()`. This information can be exploited by `potCreator` by inserting an address it control at that specific index in `participants` array.

Recommendation: This is related to the difficulty of having true onchain randomness. Since Blast is an OP fork, consider investigating `block.prevrandao` as a source of randomness. The basic requirement here is that the random number should be unpredictable by any entity before `rotateLiquidity()` is executed.

3.2 High risk

3.2.1 Subsequent rounds of winner selection is blocked for a pot

Severity: High risk

Context: QuailFinance.sol#L145-L146

Description: Second call to `rotateLiquidity()` for a pot reverts due to the call to `entropy.reveal()`. Once a `sequenceNumber` is used, entropy reverts if it's used again as it clears the request. Hence, a pot cannot be used to select a winner again.

Recommendation: Consider selecting a new `sequenceNumber` for each round of winner selection. Another solution can be to discard the concept of rounds for a pot, choose a winner only once, and that pot cannot be joined again.

3.2.2 `potCreator` can block liquidity rotation

Severity: High risk

Context: QuailFinance.sol#L142-L142

Description: Only `potCreator` can call `rotateLiquidity()`. If `potCreator` for any reason doesn't call this function, funds are permanently locked. The only way to withdraw it would be to use the merkle tree which we have recommended to change to so that it can't access user funds.

`potCreator` may not call `rotateLiquidity()` for multiple reasons like losing the private key, or maliciousness. This `require` check can't be removed as only the requestor of the random number in `createPot()` should reveal the random number used to declare winner, otherwise the call to `entropy.reveal()` reverts.

Recommendation: Consider adding a time duration after which `rotateLiquidity()` cannot be called and participants can withdraw their deposits.

3.2.3 `potCreator` can participate in multiple rounds and lose rewards

Severity: High risk

Context: QuailFinance.sol#L169-L170

Description: The intention of the team is to not let the winner enter the round again. However, the `potCreator` always becomes participant in the next round. Hence, if `potCreator` is a winner, it breaks assumptions taken for the winner that an address wins only once in a pot. If `potCreator` wins again, the reward won in a future round overwrites all the rewards won in past rounds making it unclaimable.

Recommendation: Consider checking `pot.hasWon` to ensure `potCreator` is only added if it hasn't won before.

3.2.4 Signature can be reused

Severity: High risk

Context: QuailFinance.sol#L127-L128

Description: An authenticated address (say `A`) can join multiple times due to signature replay attack. The nonce used here also comes as an argument, hence can be reused. This has two issues:

- It blocks other authenticated addresses to participate.
- `A` can enter into the pot so many times that it becomes the majority in participants (up to `numParticipants - 1` times). So with very high probability, it wins the pot effectively guaranteeing winning and taking other participants' contributions.

Recommendation: Ensure that each authenticated address joins a round at most once.

3.2.5 `createPot()` reverts due to incorrect condition on fee

Severity: High risk

Context: QuailFinance.sol#L90-L90, QuailFinance.sol#L97-L97

Description: `entropy` needs `fee` amount to be sent as `msg.value`, but `createPot()` doesn't let the caller send `fee` value along with the call due to this `require` condition:

```
require(msg.value < fee, "Insufficient fee");
```

Thus, the call reverts blocking the creation of a pot.

Recommendation: Update this `require` condition to `msg.value == fee`. You can also consider `msg.value >= fee` but this can lead to over-charging user.

3.2.6 `riskPoolBalance` is overwritten

Severity: High risk

Context: QuailFinance.sol#L160-L160

Description: `riskPoolBalance` is reassigned instead of adding it to the new amount:

```
pot.riskPoolBalance = riskPoolBalance;
```

`riskPoolBalance` currently is locked but the intention is to reuse it in later rounds if a player doesn't join. In that case, the aggregated risk pool won't be used since it only stores the latest value.

Recommendation: Add the new value instead of overwriting;

```
pot.riskPoolBalance += riskPoolBalance;
```

3.2.7 `rotateLiquidity()` can revert if all participants don't join a pot

Severity: High risk

Context: QuailFinance.sol#L146-L146

Description: It's possible all joiners don't join a pot. So `pot.participants.length < pot.numParticipants` which can make `winnerIndex` go out of bounds:

```
uint256 winnerIndex = uint256(randomNumber) % pot.numParticipants;
address winner = pot.participants[winnerIndex];
```

Recommendation: Consider selecting `winnerIndex` as follows:

```
uint256 winnerIndex = uint256(randomNumber) % pot.participants.length;
```

3.2.8 User funds can be withdraw without their permission

Severity: High risk

Context: QuailFinance.sol#L229-L229

Description: `claimFunds()` enables the owner to withdraw all USDB funds in the contract. So pot participants funds can be withdrawn without their permission:

- Owner creates a merkle tree with leaves such that the total claimAmount in leaves equal to the deposits made by pot participants.
- `claimFunds()` is called and withdraws all funds.

Recommendation: Make sure programmatically that only the yield can be claimed and nothing more.

3.2.9 Claimed gas is locked forever

Severity: High risk

Context: QuailFinance.sol#L213-L213

Description: `claimMyContractGas()` claims gas accumulated by the contract and deposits it into the contract itself. There is no functionality to withdraw this claimed amount locking it forever.

Recommendation: Consider who should be able to ultimately claim the allocated gas and implement a withdraw feature accordingly.

3.3 Medium risk

3.3.1 `withdrawRevenue()` transfers USDB back to itself

Severity: Medium risk

Context: QuailFinance.sol#L207-L208

Description: `withdrawRevenue()` transfers USDB back to itself, thus making this function unnecessary since the funds are already in the contract. The intention here is to withdraw it to owner's address or a specified address as mentioned in comments.

Recommendation: Consider withdrawing USDB owner's address or add a `receiver` argument and transfer USDB to that address.

3.3.2 `riskPoolBalance` is not used

Severity: Medium risk

Context: QuailFinance.sol#L160-L160

Description: For each winner picking, a portion of the funds is taken to account for risk balance. This balance is never used again, and the only way to retrieve is for the owner to include it in the merkle tree which is not the intended use for the risk pool.

Recommendation: Consider removing risk pool or completing the feature for which risk pool was introduced.

3.3.3 Upgradable contracts used for immutable deployment

Severity: Medium risk

Context: QuailFinance.sol#L14-L14

Description: Upgradable contracts are used but no Proxy contract found. Upgradable contracts are designed to be used with Proxy contracts due to the intricacies of `delegatecall` and storage slot clashing concerns. The repository doesn't contain any proxy contract indicating that the contracts will be deployed in an immutable way.

Recommendation: Consider using the simple `Ownable` contract instead of `OwnableUpgradable`, remove all Upgradable inherited contracts, remove `initializer` function and adjust it in the constructor.

If you want to use proxy, then update the project to use a proxy contract.

3.3.4 No smart contract dev framework used

Severity: Medium risk

Context: QuailFinance.sol#L14-L14

Description: Frameworks like Foundry lets you write unit tests, do fork testing and write deployment scripts based on which chain you're deploying on (Blast testnet vs mainnet here for example). Some bugs reported in this report would have been caught through unit and fork testing.

Recommendation: Make this a Foundry project and add unit and fork tests.

3.4 Low risk

3.4.1 `contributions` doesn't have any effect

Severity: Low risk

Context: QuailFinance.sol#L42-L42, QuailFinance.sol#L60-L60

Description: `contributions` mapping isn't used to enforce any requirement on the contract. It's only assigned in `joinPot()` and `createPot()`. The assignment is missed in `rotateLiquidity()`. As per comments, it's supposed to prevent multiple deposits from the same address.

Recommendation: Consider the intended use of `contributions` and complete the feature which is supposed to use this variable.

3.4.2 Modulo bias

Severity: Low risk

Context: QuailFinance.sol#L146-L146

Description: `randomNumber` is in the range $[0, 2^{256}-1]$, it is then mapped to the range $[0, \text{pot.numParticipants}-1]$. This introduces modulo bias such that some range of indices in the end of the `participants` array are less likely to be picked compared to the indices in the beginning of the array. This makes the winner selection non-uniform and biased towards early joiners of the round.

Recommendation: There is no good solution to avoid this bias onchain. We recommend informing users of this bias.

3.4.3 Use domain separator for message hashing

Severity: Low risk

Context: QuailFinance.sol#L127-L127

Description: Currently, the signatures can potentially be replayed on different smart contracts and different chains. This is because domain separator isn't used as specified in EIP-712. A domain separator defines the verifying contract and chain ID which is then included in the message to be hashed. This restricts the surface area of a signature replay attack.

Recommendation: Add domain separator in the message being hashed.

3.4.4 Make `joinPot()` non-payable

Severity: Low risk

Context: QuailFinance.sol#L124-L124

Description: `joinPot()` doesn't use any eth sent to it, so it can be made non-payable.

Recommendation: Delete `payable` keyword for `joinPot()`.

3.4.5 Mapping and internal function never used

Severity: Low risk

Context: QuailFinance.sol#L185-L185

Description: `userYield` mapping and `updateUserYield` function is never used as it's an `internal` function.

Recommendation: Consider if they were introduced for any feature. If not, delete the related code.

3.5 Gas Optimization

3.5.1 Unnecessary manipulation of `pot.participants` before deletion

Severity: Gas Optimization

Context: QuailFinance.sol#L150-L154, QuailFinance.sol#L168-L168

Description: `rotateLiquidity()` removes the winner from `pot.participants` and then deletes the array. Hence, manipulating the array to remove the winner is unnecessary.

Recommendation: Do no manipulate the array before deleting it.

3.5.2 Cache storage variable

Severity: Gas Optimization

Context: QuailFinance.sol#L150-L152, QuailFinance.sol#L176-L179

Description: `pot.amountWon[msg.sender]` and `pots[_potId].participants.length` is read twice. Caching them results in saving one storage read.

Recommendation: Cache this data in variables to avoid reading from storage multiple times.

3.6 Informational

3.6.1 Majority of participants can collude

Severity: Informational

Context: Global scope

Description: If majority of addresses (up to 1 less than the maximum number of participants) join a round, they have a very high likelihood of winning. This may be a strategy to guarantee a win with high probability.

However, from an individual player's perspective, they still have the same chance of winning (`1/N`) if uniform distribution over all the players is assumed. So even if majority participants collude, it doesn't change the risk/reward ration of an individual player.

Recommendation: This is just for informational purpose to make the Quail team aware. As mentioned, this cannot be mitigated and the probability of winning of an individual player doesn't change.

3.6.2 Consider using 2 step update for owner and admin

Severity: Informational

Context: QuailFinance.sol#L225-L227

Description: Single-step change of owner or admin poses a security risk. This approach is not recommended, as if an incorrect address is mistakenly set by the owner, it would render the owner and admin specific features inaccessible.

Recommendation: Consider a two-step approach when changing privileged roles:

- The current owner proposes a new address for the ownership change.
- In a separate transaction, the proposed new address can then claim the ownership.

An example implementation of the mentioned pattern is `Ownable2Step` by `OpenZeppelin`.

3.6.3 Careful with merkle tree management

Severity: Informational

Context: QuailFinance.sol#L229-L240

Description: Merkle tree claim process needs to be handled with care. If the owner doesn't prepare the merkle tree properly, it can lead to some claims to be blocked. In particular:

- Make sure that each address appears only once in the tree. If it occurs more than once, then `claimFunds()` cannot handle it as it takes `claimAmount` as an argument which is stored in the leaf but only transfers the difference of `claimAmount` and the previous claimed amount. So the full amount is not transferred.
- The owner also needs to update the root regularly as new USDB yield from Blast accrues. Make sure that the claim amount embedded in a leaf is a cumulative amount irrespective of how much has been claimed by an address before. This is important as `claimFunds()` only transfers the difference of the leaf amount and the previously claimed amount.

Recommendation: Make sure to account for all the behaviors when constructing and updating the merkle root.

3.6.4 `RotationCompleted` event emits the next round number

Severity: Informational

Context: QuailFinance.sol#L166-L166, QuailFinance.sol#L171-L171

Description: `pot.currentRound` is incremented before `RotationCompleted` event is emitted. Hence, the value emitted for `round` here is indicating towards the next round which has just begun. However, for the last round, it emits the current round number which is inconsistent.

Recommendation: Consider subtracting `1` from `pot.currentRound` before emitting for all rounds except the last.

3.6.5 Boolean NOT operator can be used

Severity: Informational

Context: QuailFinance.sol#L132-L132

Description: You can refactor the following to check if the boolean is `false` in a more canonical way by using the NOT (`!`) operator.

Recommendation: Consider the following:

```
!pot.hasWon[msg.sender]
```

3.6.6 Several important TODOs left

Severity: Informational

Context: QuailFinance.sol#L123-L123

Description: Several TODOs mentioned in the code which point to missing code implementation.

Recommendation: Consider each TODO for implementation.