Lizard Staking Security Review

Pashov Audit Group

Conducted by: pashov

March 27th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Lizard Staking** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Lizard Staking

The **Lizard Staking** protocol allows holders of `Ethlizards` and `Genesis Ethlizards` NFTs to stake them in exchange for rewards in the form of `USDC`. Staking an NFT will mint you a `Locked Lizards` NFT and lock your `(Genesis) Ethlizards` NFT in the contract for 90 days during which it can't be withdrawn, but accrued rewards can be claimed any time while you are still staking. The protocol uses a shares-based approach, where the rewards are in a "pool" and each staker owns a share of this pool. There is also a shares rebasing (inflation) mechanism based on time staked, as well as a reset mechanism for those rebases.

Unexpected/Interesting Design choices

Approvals for the `Locked Lizards` NFTs are constrained by the `onlyApprovedContracts` modifier which implements a whitelist on contracts that can be approved.

Withdrawing your staked Lizard NFT will not burn your `Locked Lizards` - it stays in the staking contract and if you re-stake you get it transferred back to you.

There is a `retractLockedLizard` functionality, which allows a staker to forcefully transfer the `Locked Lizards` NFT back to his address if it was transferred to another one.

If a user does not call `claimReward` before he calls `withdrawStake` then this will result in the accrued rewards being locked up in the staking contract forever, without a way for the user or the protocol owner to claim/rescue them.

More docs

Ethlizards Collection & Genesis Ethlizards Collection

Threat Model

Roles & Actors

- Owner - can control the allowed contracts to be approved, the reset share value, the min reset value, the council address for depositing rewards and can switch on and off the deposits and reset the `startTimestamp` and `lastGlobalUpdate`,
- Staker - a user that stakes an `Ethlizard` or a `Genesis Ethlizard` NFT with the goal of receiving `USDC` rewards
- Original `Locked Lizards` owner - can forcefully transfer a `Locked Lizards` NFT back to himself
- Allowed contracts - addresses of contracts (can be EOAs too) that can be set as operators for `Locked Lizards` NFTs
- Council - can deposit rewards and trigger a new pool creation

Security Interview

Q: What in the protocol has value in the market?

A: The actual staked Lizard NFTs and the `USDC` used to distribute reward.

Q: What is the worst thing that can happen to the protocol?

A: An attacker stealing (partially or fully) the `USDC` balance of the contract or the staked NFTs into it or putting the contract in a state of DoS.

Q: In what case can the protocol/users lose money?

A: In the case when they can't `withdraw` their staked NFTs or `claim` their accrued rewards.

Potential attacker's goals

- Put the system in a state of DoS, where each `claimRewards` or `withdrawStake` transaction reverts, executing a griefing attack on stakers
- Steal the `USDC` rewards in the pool
- Steal the NFTs staked in the pool
- Take advantage of a rewards calculation bug and receive more reward than expected

Potential ways for the attacker to achieve his goals

- Staking an NFT and then calling `claimRewards` in a specific time with a specific `_poolNumber` argument
- Claiming rewards multiple times when he should have been able to claim once
- Call the `withdrawStake` function in a way that will transfer another staker's NFT to him
- Cheat the `depositStake` method into thinking he staked when he didn't

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - 448b8d934a0591e10aea871d4a405e3fa7aa28c4**

Scope

The following smart contracts were in scope of the audit:

- `LizardLounge`
- `interfaces/**`

7. Executive Summary

Over the course of the security review, pashov engaged with Lizard Staking to review Lizard Staking. In this period of time a total of **29** issues were uncovered.

Protocol Summary

Protocol Name	Lizard Staking
Date	March 27th, 2023

Findings Count

Severity	**Amount**
Critical	2
High	2
Medium	4
Low	6
QA	15
Total Findings	**29**

Summary of Findings

ID	Title	Severity	Status
[C-01]	It's impossible for a user to claim his rewards, as claimReward will never send out USDC	Critical	Resolved
[C-02]	Calculation for owedAmount will round down to zero	Critical	Resolved
[H-01]	Users will forever lose their accrued rewards if they call withdrawStake before calling claimReward first	High	Resolved
[H-02]	Wrong check in claimCalculation will result in less rewards received for users	High	Resolved
[M-01]	Looping over unbounded array can result in a state of DoS	Medium	Resolved
[M-02]	Missing constraint on the setter method of a percentage value	Medium	Resolved
[M-03]	Owner has the power to zero-out user's daily interest on rewards	Medium	Resolved
[M-04]	Constraining approvals only partially limits the NFTs from being sold	Medium	Resolved
[L-01]	Functionality from docs is missing	Low	Resolved
[L-02]	Implementation is not making use of ERC721's burn method	Low	Resolved
[L-03]	Unexpected behavior if user stakes in the same block as when the first pool is created	Low	Resolved
[L-04]	Division before multiplication in the calculateShareFromTime method	Low	Resolved
[L-05]	Use a two-step ownership transfer approach	Low	Resolved

[L-06]	Code naming gives an assumption that is not enforced	Low	Resolved
[QA-01]	No need to use safeTransfer since contracts are not allowed	QA	Resolved
[QA-02]	Overcomplicated math calculations	QA	Resolved
[QA-03]	Method and storage variables can be removed	QA	Resolved
[QA-04]	Open TODOs in the codebase	QA	Resolved
[QA-05]	Filename and interface name mismatch	QA	Resolved
[QA-06]	Contract is using both custom errors and require statements	QA	Resolved
[QA-07]	Typos, grammatical errors, redundancies and complexity in NatSpec docs and comments	QA	Resolved
[QA-08]	LockedLizardMinted event emission should happen in the mint method	QA	Resolved
[QA-09]	Use the delete keyword instead of assigning the default value of variables	QA	Resolved
[QA-10]	Variables can be turned into an immutable or a constant	QA	Resolved
[QA-11]	Most state-changing methods do not emit events	QA	Resolved
[QA-12]	Interface is not needed	QA	Resolved
[QA-13]	Make isLizardWithdrawable directly return the result of the check	QA	Resolved
[QA-14]	Naming problem in onlyApprovedContracts	QA	Resolved
[QA-15]	The IERC721 interface is not needed	QA	Resolved

8. Findings

8.1. Critical Findings

[C-01] It's impossible for a user to claim his rewards, as `claimReward` will never send out `USDC`

Severity

Impact: High, because users will never receive rewards from the contract

Likelihood: High, because the code just uses the ERC20 API incorrectly

Description

The `claimReward` method should be used by a staker to receive `USDC` rewards for his locked NFTs. This won't ever work, as the transfer of the rewards is implemented with this code:

```
USDc.transferFrom(msg.sender, address(this), claimableRewards);
```

This is wrong as it will transfer `USDC` from the staker to the staking contract instead of the other way around.

Recommendations

Change the code in the following way:

```
- USDc.transferFrom(msg.sender, address(this), claimableRewards);
+ USDc.transfer(msg.sender, claimableRewards);
```

Make sure to always use the ERC20 API correctly and also to have complete code coverage with unit tests of the codebase prior to having an audit.

[C-02] Calculation for `owedAmount` will round down to zero

Severity

Impact: High, as this will result in 0 claimable rewards for users when they should have been able to claim some

Likelihood: High, as this will happen any time the user's share is smaller than the pool's cached global share, which is almost always

Description

The `claimCalculation` method calculates the `owedAmount` that is about to be send to the user in the form of `USDC` rewards with the following calculation:

```
owedAmount =
  (currentShareRaw / pool[_poolNumber].currentGlobalShare) * pool[_poolNumber].value;
```

This happens both if the `_poolNumber == 1` and if it is a different value, but the code is present in both cases. The issue is that it does division before multiplication, where if the `pool[_poolNumber].currentGlobalShare` value is bigger than the `currentShareRaw` value, the division will round down to zero resulting in zero `owedAmount`. This will almost always happen as it is expected that the pool's cached `currentGlobalShare` will be bigger than a single user's raw share. This means that no matter how much a user waits he won't be able to claim his rewards for this pool, leaving them stuck in the contract forever.

This issue was partly noticed by the developer mid-audit, where he fixed one of the places where `owedAmount` was calculated, but the other `owedAmount` calculation error one wasn't discovered.

Recommendations

Change the code in the following way:

```
- owedAmount =
- (currentShareRaw / pool[_poolNumber].currentGlobalShare) * pool[_poolNumber].value;
+ owedAmount = currentShareRaw * pool[_poolNumber].value
+     / pool[_poolNumber].currentGlobalShare;
```

So this way you do multiplication before division which will not round down to zero, as the pool's value is in `USDC` that has 6 decimals, but the shares have 18 decimals.

8.2. High Findings

[H-01] Users will forever lose their accrued rewards if they call `withdrawStake` before calling `claimReward` first

Severity

Impact: High, as this will lead to a monetary loss for users

Likelihood: Medium, as even though the front-end will enforce the right sequence of calls, the Gitbook docs falsely claims re-staking will re-gain user's access to their rewards

Description

The contract is implemented so that if a user calls `withdrawStake` without first calling `claimReward` for each reward pool then the staker will lose all of his unclaimed rewards forever, they will be locked into the staking contract. While the front-end will enforce the right sequence of calls, the Gitbook docs state that `When un-staked, a user will lose access to all their pending rewards and lose access to future rewards (unless they re-stake)` which gives the impression that you can re-stake and then you will re-gain access to your unclaimed rewards, but this is not the case as the `withdrawStake` method removes the data needed for previous rewards calculation.

Since the docs give a misleading information about they way this mechanism works and also users can interact directly with the smart contract in a bad way for them (when they are not malicious) this has a higher likelihood of happening and resulting a monetary value loss for users.

Recommendations

One possible solution is to enforce zero unclaimed rewards when a call to `withdrawStake` is made by reverting if there are any such unclaimed rewards. Another one is to just call `claimReward` in `withdrawStake`.

[H-02] Wrong check in `claimCalculation` will result in less rewards received for users

Severity

Impact: High, as this will lead to a monetary loss for users

Likelihood: Medium, as it happens only for the pool with an ID of 1

Description

The first `if` statement in `claimCalculation` checks `if (_poolNumber == 1)` and does not factor in any inflation for that particular pool. The problem is that (it is also explained in the comment above the `if` statement) the intention was to check if there was only 1 pool (or if it was the first pool) then there is no need to do inflation calculations, which result in a higher reward. But when you have `_poolNumber == 1` this means that you have at least 2 pools, as arrays start from an index of 0, so 1 is actually for the second pool in the `pool` array. This will result in all claimers of the rewards for staking in the pool with an ID of 1 missing out on their inflation rewards.

Recommendations

Change the code in the following way:

```
- if (_poolNumber == 1) {
+ if (_poolNumber == 0) {
```

8.3. Medium Findings

[M-01] Looping over unbounded array can result in a state of DoS

Severity

Impact: High, as the contract will be in a state of DoS, without a way for anyone to withdraw NFTs or claim rewards

Likelihood: Low, as it requires a lot of pools added or a malicious owner

Description

The `claimCalculation` and `getCurrentShareRaw` methods both loop over the `pool` array to do proper calculations. The problems is that there is no way to pop elements out of the array, but there is no upper bound on the length of the array. Each time the `currentRewards` are more than or equal to the `minResetValue`, the `createPool` method will be called, adding a new element to the `pool` array. If at some point there are now a large number of pools, iterating over them will become very costly and can result in a gas cost that is over the block gas limit. This will mean that a transaction cannot be executed anymore, leaving the contract's main functionalities (withdrawing the staked NFTs and claiming rewards) in a state of DoS.

Recommendations

Limit the number of pools that can be created, for example a maximum of 25 pools created.

[M-02] Missing constraint on the setter method of a percentage value

Severity

Impact: High, as it will result in wrong reward calculations

Likelihood: Low, as it requires a malicious/compromised owner or a big error on his side

Description

The `setResetShareValue` lacks a check that the `_newShareResetValue` argument is not more than 100%. Since it is expected that the value will be in percentages, setting a value that is bigger than 100 will mess with the important calculations in the contract, one of which is the rewards to claim calculation. This can make users receive a smaller reward than what they have earned since a bigger `resetShareValue` equals smaller rewards for users.

Recommendations

Add a check in `setResetShareValue` that the `_newShareResetValue` argument is not more than 100%.

[M-03] Owner has the power to zero-out user's daily interest on rewards

Severity

Impact: High, as users can lose their right to claim accrued rewards

Likelihood: Low, as it requires a malicious/compromised owner

Description

The `setDepositsActive` method resets `startTimestamp` and `lastGlobalUpdate`. The owner can front-run each `claimReward` transaction and by resetting the `startTimestamp` this will result in 0 `requiredRebases` in `calculateShareFromTime`, so the user will lose on his daily interest. On the other side, by resetting `lastGlobalUpdate` this will make `updateGlobalShares` never do a rebase, which will never inflate the `overallShare` which also shouldn't be possible.

Recommendations

Make the `setDepositsActive` method callable only once after contract deployment.

[M-04] Constraining approvals only partially limits the NFTs from being sold

Severity

Impact: High, as it can lead to scams and bugs when integrating with other games/protocols

Likelihood: Low, as such sales or integrations are not currently expected to happen and because information about this is present in the docs

Description

Constraints on approvals (the `onlyApprovedContracts` modifier) were added so that the `Locked Lizards` NFTs can't be sold in marketplaces like OpenSea, Blur etc. This only partially limits selling the NFTs because users can always do OTC trades. Those trades will be scams though, since the original NFT owner can call `retractLockedLizard` anytime and re-gain ownership of the NFT. Not only sales will be problematic, but for example integrations with NFT games - the games are not expected to work properly with NFTs that can be retracted, as this opens up multiple attack-vectors.

Recommendations

Either remove the `onlyApprovedContracts` modifier and allow sales and integrations by removing the `retractLockedLizard` functionality, or just forbid the `approve` and `transfer` functionality altogether as otherwise they can result in problems.

8.4. Low Findings

[L-01] Functionality from docs is missing

The Gitbook says: `In order to claim rewards, the user will also need to engage in at least one on-chain governance vote/action with their Ethlizards, to ensure active participation.`. This functionality is not present in the contract and can lead to false assumptions from users/protocol devs. Either remove it from the docs or update the code by implementing it.

[L-02] Implementation is not making use of ERC721's `burn` method

The current implementation stores the LLZ NFT after a user withdraws his stake and it transfers it back to him on a subsequent deposit. It would be better if you burn the LLZ NFT on a withdraw and then re-mint it on subsequent deposit as this follows the usual best-practice pattern related to ERC721 NFTs, while the currently used one is error-prone.

[L-03] Unexpected behavior if user stakes in the same block as when the first pool is created

If there is only 1 rewards pool and a user has staked in exactly the same block as when the pool was created, then both

```
(pool[pool.length - 1].time) <= timeLizardLocked[_tokenId]
```

and

```
timeLizardLocked[_tokenId] <= pool[0].time
```

will return `true`. The problem is the first check is in the `if` of `getCurrentShareRaw`, while the second one is in the `else if`, and depending on which branch the code takes different calculations happen for the share amount. Removing the `=` sign from either of them will fix the issue, where I would say it is more correct to remove it in the `if` statement, as it is fair that a user that staked in the same block gets the shares inflation of the pool.

There is also another problem that is very close to this one: in `getCurrentShareRaw` if a user stakes in the same `block.timestamp` as when the first pool is created, then his share will be calculated as if he is included in that first pool. This is not the case for the `claimCalculation` function, where if the user has staked in the same block (same `block.timestamp`) where the first pool was created, his rewards won't include the rewards from the first block. This is unexpected as the protocol doesn't document this behavior is intended - receiving the pool's share inflation but not receiving the pool's rewards when you stake in the same block as when the first pool was created.

My recommendation is that the user should receive both inflation and rewards for his stake if he staked in the same block as when a pool was created, think through this in depth.

[L-04] Division before multiplication in the `calculateShareFromTime` method

The `requiredRebases` variable is calculated by using division by `1 days`. It can unexpectedly round down to zero but this won't lead to any problems, as then the result is passed to the `calculateRebasePercentage` method, where it is used as a "power of" value, so then the `calculateRebasePercentage` will return 1 even if it received 0 as an argument. This should be well documented and understood by developers and auditors. Add proper comments in the code.

[L-05] Use a two-step ownership transfer approach

There are 4 method with the `onlyOwner` modifier which shows that the `owner` role is an important one. Make sure to use a two-step ownership transfer approach by using `Ownable2Step` from OpenZeppelin as opposed to `Ownable`

as it gives you the security of not unintentionally sending the `owner` role to an address you do not control.

[L-06] Code naming gives an assumption that is not enforced

The `allowedContracts` mapping can contain EOAs as well as contracts in it. Another dev can expect only contracts to be able to call methods with the `onlyApprovedContracts` modifier, but that is not the case. Fix this by ensuring every allowed address is a contract address by adding a check that the codesize in the address is > 0 when setting it in `setAllowedContracts`.

8.5. QA Findings

[QA-01] No need to use `safeTransfer` since contracts are not allowed

The `depositStake` method disallows contract calling it (even though code is commented out, it says it will be uncommented out) so you never need to do use the ERC721 `_safeTransfer` functionality since it is always done to the initial depositor. Use the normal `_transfer` functionality instead.

[QA-02] Overcomplicated math calculations

The following code from `calculateShareFromTime` is overcomplicated and can be simplified in the following way:

```
- uint256 requiredRebases = ((_currentTime - startTimestamp) -
- (_previousTime - startTimestamp)) / 1 days;
+ uint256 requiredRebases = (_currentTime - _previousTime) / 1 days;
```

[QA-03] Method and storage variables can be removed

Rename `stakePoolClaims` to `rewardsClaimed`, remove `isRewardsClaimed` and then just use the automatically generated getter of it for simplicity. This will also result in a gas optimization. Also, the `resetCounter` and `rebaseCounter` storage variables are not read on-chain so you can just emit events on `reset` or `rebase` and do the counting off-chain.

[QA-04] Open `TODO`s in the codebase

There are currently 4 opened `TODO`s in the code, one of which shows an intent to add code/features post-audit - `// TODO: ADD A MERKLE SIGNATURE HERE ONCE FRONTEND IS FINALISED`. Remove or resolve all of them.

[QA-05] Filename and interface name mismatch

In both `IEthLizards` and `IGenesisEthLizards` there is a mismatch between the filenames and the interface names - while the filenames write `Lizards` with a capital `L`, the interfaces use a lower-case one. Same problem is present for the `IUSDc` interface that is contained in the `IUSDC` file. Make sure to be consistent in the naming as this can lead to subtle errors.

[QA-06] Contract is using both custom errors and `require` statements

Make sure the contract consistently uses custom errors everywhere as it is more gas efficient and helps with the interoperability of the protocol.

[QA-07] Typos, grammatical errors, redundancies and complexity in NatSpec docs and comments

There are multiple problems in the NatSpec docs and comments of the `LizardLounge` contract:

- The `IUSDC` file has this `// Unit testing for the LizardLounge Contract` comment, which implies interface is only used for testing but it is used in the `LizardLounge` contract as well
- The NatSpec of `isLizardWithdrawable` says that it checks if a lizard is transferrable but it actually checks if it is withdrawable
- `Checks if the rewards of a lizard for a specific pool has been claimed` -> `Checks if the rewards of a lizard for a specific pool have been claimed`
- Strange comment in `calculateRebasePercentage` - `// Do not times`, should be removed or updated
- Incomplete sentence in the NatSpec of `calculateRebasePercentage` - `@notice We calculate the 1.005^_requiredRebases and`, complete the sentence
- The NatSpec and comments in `calculateRebasePercentage` mention some technical documents but there is no link to them - add it
- Incomplete sentences in the NatSpec of `depositStake` - update it so it is correct
- NatSpec of `depositStake` says `Allows user to deposit their regular Ethlizards for staking` but it allows `Genesis Ethlizards` staking as well, update it
- Typos: `firsts` -> `first`, `depositer` -> `depositor`, `CallerNotDepositer` -> `CallerNotDepositor`, `everytime` -> `every time`
- The NatSpec of `updateGlobalShares` says the method `Gets the current global share counter` which is incorrect, update it so it is correct
- Grammatical errors:
  - `TokenId where share is being calculated` -> `TokenId for which share is being calculated`
  - `Transfer the user the USDC rewards` -> `Transfer the USDC rewards to the user`
  - `If the no pools have been created` -> `If no pools have been created`
  - `after the user is staked` -> `after the user has staked`
  - `// First time stakers mints their...` -> `// First time stakers mint their ...`
  - `A lizard is transferrable if it been over 90 days since it was deposited` -> `A lizard is transferrable if more than 90 days have passed since it was deposited`
  - `... user are protected ...` -> `... users are protected ...`

[QA-08] `LockedLizardMinted` event emission should happen in the mint method

Move the emission of the `LockedLizardMinted` event to the `mintLLZ` method as it is emitted only when the method is called.

[QA-09] Use the `delete` keyword instead of assigning the default value of variables

In `withdrawStake the` timeLizardLocked `and` originalLockedLizardOwners `for a staked NFT are reset by assigning them to 0 or` address(0) `. It is a best practice is to just use the` delete` keyword instead, there is no need to manually assign the type's default values.

[QA-10] Variables can be turned into an `immutable` or a `constant`

The `nominator` variable's value is known at compile-time so you can make it `private constant` as it is also not expected to be called outside of the contract, while the `Ethlizards`, `GenesisLiz` and `USDc` variables can be made `immutable` since they are only set in the constructor and never changed after that.

[QA-11] Most state-changing methods do not emit events

Examples for this are `setDepositsActive` or `setCouncilAddress` - state-changing methods should emit events so that off-chain monitoring can be implemented. Make sure to emit a proper event in each state-changing method to follow best practices.

[QA-12] Interface is not needed

The `IUSDc` interface is not needed in the codebase - import the OpenZeppelin `IERC20` interface and use it instead. Also remove the `ABDKMath64x64` from the codebase and just import it as an external dependency, as there is no need to keep it there.

[QA-13] Make `isLizardWithdrawable` directly return the result of the check

There is no need to have an if-else statement in the method, instead just directly do:

```
return block.timestamp - timeLizardLocked[_tokenId] >= 90 days;
```

in `isLizardWithdrawable`.

[QA-14] Naming problem in `onlyApprovedContracts`

The `onlyApprovedContracts` modifier uses three different words for the same thing - `approved`, `allowed` and `whitelisted`. Stay consistent and use only one word for one meaning in the context of the protocol, for example `whitelisted`.

[QA-15] The `IERC721` interface is not needed

The `IEthlizards` interface imports and inherits from the `IERC721` interface - this is not needed as none of its methods are used. Remove the `IERC721` inteface and its import.