Zerem 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 **Zerem** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Zerem

The code was reviewed for a total of 10 hours.

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* - 667a41b577647f0d95591a5f9928a43b976b8e25**

Scope: `Zerem.sol`

7. Executive Summary

Over the course of the security review, pashov engaged with Zerem to review Zerem. In this period of time a total of **20** issues were uncovered.

Protocol Summary

Protocol Name	Zerem
Date	March 27th, 2023

Findings Count

Severity	**Amount**
High	1
Medium	7
QA	12
Total Findings	**20**

Summary of Findings

ID	Title	Severity	Status
[H-01]	The unlockExponent does not work as intended when it is ≠ 1	High	Resolved
[M-01]	Unsafe call to ERC20::transfer can result in stuck funds	Medium	Resolved
[M-02]	Gas stipend for external call might be insufficient and lead to stuck ETH	Medium	Resolved
[M-03]	Gas griefing/theft is possible on unsafe external call	Medium	Resolved
[M-04]	Centralisation risk with liquidationResolver as it can steal 100% of locked funds	Medium	Resolved
[M-05]	Missing configuration validations & constraints can lead to stuck funds	Medium	Resolved
[M-06]	ERC20 tokens that have a fee-on-transfer mechanism require special handling	Medium	Resolved
[M-07]	Protocol does not work with ERC20 tokens that have a mechanism for balance modifications outside of transfers	Medium	Resolved
[QA-01]	Use latest Solidity version with a stable pragma statement	QA	Resolved
[QA-02]	Add NatSpec documentation	QA	Resolved
[QA-03]	Code is not formatted properly	QA	Resolved
[QA-04]	Move event emissions to the methods where the action happens	QA	Resolved
[QA-05]	Typos in comments	QA	Resolved
[QA-06]	Missing non-zero address checks in the	QA	Resolved

	constructor		
[QA-07]	No need to cast to address payable in _sendFunds	QA	Resolved
[QA-08]	Some functions can be fused into one	QA	Resolved
[QA-09]	Open TODO in the code	QA	Resolved
[QA-10]	Remove duplicated code	QA	Resolved
[QA-11]	Update external dependency to latest version	QA	Resolved
[QA-12]	Consider using a differet key in the pendingTransfers mapping	QA	Resolved

8. Findings

8.1. High Findings

[H-01] The `unlockExponent` does not work as intended when it is ≠ 1

Proof of Concept

The documentation and the chart in `README.md` shows that it is expected that when `unlockExponent == 0` then immediately after funds `unlockDelaySec` the user can claim his whole locked amount. This is actually not working as intended, let's look at the `_getWithdrawableAmount` function:

1. To calculate the amount to unlock, we have the following: `uint256 totalUnlockedAmount = (record.totalAmount * factor) / precision;`
2. `record.totalAmount` is the total locked amount, `precision` is a constant with a value of `1e8` and `factor` is calculated by this:

```
uint256 factor = deltaTimeNormalized ** unlockExponent;

if (factor > precision) {
        factor = precision;
}
```

1. If we have `unlockExponent == 0` then `factor` is always equal to 1, which is less than `1e8` so `factor == 1`
2. Now if we go back to the total amount to unlock math, we will get `uint256 totalUnlockedAmount = (record.totalAmount * factor) / precision;` so `uint256 totalUnlockedAmount = record.totalAmount / precision`

The expected unlocked amount was equal to `record.totalAmount` but instead we got `record.totalAmount / precision` which is incorrect. Now every subsequent time the `_getWithdrawableAmount` function is called, the math will be the same and the code will basically think there is no newly unlocked amount. This means that no user that has locked funds in Zerem will be able to

withdraw more than `totalLockedAmount / 1e8` ever, all of the other tokens will be stuck.

There is also a problem when `unlockExponent > 1` , because the computed `factor` can easily be `>= precision` which will result in 100% of funds being unlocked too early.

Here is the important math:

```
uint256 deltaTimeNormalized = (deltaTimeDelayed * precision) / unlockPeriodSec;

uint256 factor = deltaTimeNormalized ** unlockExponent;

if (factor > precision) {
      factor = precision;
}
uint256 totalUnlockedAmount = (record.totalAmount * factor) / precision;
```

and let's look at example scenario:

1. `unlockPeriodSec == 100 000`, `deltaTimeDelayed == 100` and `precision == 1e8` so `deltaTimeNormalized == 1e5`
2. if `unlockExponent > 1` for example when equal to 2, we will get `factor = 1e5 ** 2`, so `1e10` which is `> precision` that is `1e8` so now `factor = precision`
3. Now `totalUnlockedAmount = record.totalAmount * factor / factor` which is `record.totalAmount`
4. even though only 1/1000th of the unlock period has passed and the `unlockExponent` was just `2`, the user can already claim all of their locked tokens.

Impact

The protocol does not work as expected in its core functionality and can also result in stuck tokens (value loss) for users or tokens unlocked too early, so it is High severity.

Recommendation

Redesign the `unlockExponent` logic or just hardcode it to always be linear (a value of 1)

Client response

Resolved by removing the `unlockExponent` logic

8.2. Medium Findings

[M-01] Unsafe call to `ERC20::transfer` can result in stuck funds

Proof of Concept

In the `_sendFunds` method we have the following code for transferring ERC20 tokens

```
IERC20(underlyingToken).transfer(receiver, amount);
```

The problem is that the `transfer` function from ERC20 returns a bool to indicate if the transfer was a success or not. As there are some tokens that do not revert on failure but instead return `false` (one such example is ZRX) and also Zerem should work with all types of ERC20 tokens since it might be integrated with a protocol that does that, not checking the return value can result in tokens getting stuck. Let's look at the following scenario:

1. Alice is trying to claim some tokens from a protocol that has integrated with Zerem, so her transaction makes a call to `Zerem::transferTo`
2. The amount to claim is less than the `lockThreshold` so the code goes to the `_sendFunds` functionality directly
3. The transfer fails and since the token does not revert but returns `false` this is not accounted for by the Zerem protocol and transaction completes successfully
4. The tokens are now stuck and are not claimable by Alice anymore

Impact

If an `ERC20::transfer` call fails it will lead to stuck funds for a user. This only happens with a special class of ERC20 tokens though, so it is Medium severity.

Recommendation

Use OpenZeppelin's `SafeERC20` library and change `transfer` to `safeTransfer`

Client response

Resolved by adding `SafeERC20`

[M-02] Gas stipend for external call might be insufficient and lead to stuck ETH

Proof of Concept

The way the Zerem protocol transfers out ETH looks like this

```
payable(receiver).call{gas: 3000, value: amount}(hex"")
```

As you see, there is a gas stipend of 3000, but this might not be enough in some cases as some smart contract recipients need more than 3000 gas to receive ETH.

Examples of problematic recipients:

1. Recipient is a smart contract that has a payable fallback method which uses more than 3000 gas
2. Recipient is a smart contract that has a payable fallback function that needs less than 3000 gas but is called through a proxy, raising the call's gas usage above 3000.

Additionally, using higher than 3000 gas might be mandatory for some multi-sig wallets.

Impact

Some recipients will lose access to all of their claimable ETH from protocols that are integrated with Zerem. This requires a special type of recipient, so it is Medium severity.

Recommendation

At least doubling down the gas stipend should help in most scenarios, but maybe think about dynamic configuration options for it as well

Client response

Resolved by doubling the gas stipend

[M-03] Gas griefing/theft is possible on unsafe external call

Proof of Concept

This comment `// TODO: send relayer fees here` in the `unlockFor` method and its design show that it is possible that `unlockFor` is usually called by relayers. This opens up a new attack-vector in the contract and it is gas griefing on the ETH transfer

```
(bool success,) = payable(receiver).call{gas: 3000, value: amount}(hex"");
```

Now `(bool success, )` is actually the same as writing `(bool success, bytes memory data)` which basically means that even though the `data` is omitted it doesn't mean that the contract does not handle it. Actually, the way it works is the `bytes data` that was returned from the `receiver` will be copied to memory. Memory allocation becomes very costly if the payload is big, so this means that if a `receiver` implements a fallback function that returns a huge payload, then the `msg.sender` of the transaction, in our case the relayer, will have to pay a huge amount of gas for copying this payload to memory.

Impact

Malicious actor can launch a gas griefing attack on a relayer. Since griefing attacks have no economic incentive for the attacker and it also requires relayers it should be Medium severity.

Recommendation

Use a low-level assembly `call` since it does not automatically copy return data to memory

```
bool success;
assembly {
    success := call(3000, receiver, amount, 0, 0, 0, 0)
}
```

Client response

Resolved by using a low-level assembly `call`

[M-04] Centralisation risk with `liquidationResolver` as it can steal 100% of locked funds

Proof of Concept

Currently the `liquidationResolver` has the power to steal 100% of locked funds in the following way:

1. Call `freezeFunds` for every user that has a locked balance
2. Wait some time until the liquidation delay has passed so the `require` statement in `liquidateFunds` succeeds
3. Call `liquidateFunds` and receive all of the users' balances

This can happen if the `liquidationResolver` becomes malicious or is compromised.

Impact

Centralisation vulnerabilities usually require a malicious or a compromised account and are of Medium severity

Recommendation

Reconsider if the freeze/liquidate funds is a mandatory mechanism for the protocol

Client response

Acknowledged

[M-05] Missing configuration validations & constraints can lead to stuck funds

Proof of Concept

If a protocol integrates with Zerem it needs to deploy different instances of `Zerem.sol` for each `underlyingToken`. In the constructor there are some configurations being set but the inputs are not validated at all. Now if the deployer did not configure them correctly, or fat-fingered the deployment or if the deployment scripts were incorrect, it is possible to misconfigure the protocol in such a way that it is not obvious but leads to all locked funds getting stuck forever.

Let's look at the following scenario:

1. Mistake in the deployment script sets the `unlockDelaySec` or the `unlockPeriodSec` to be huge, or to be a concrete timestamp
2. Now in `_getWithdrawableAmount` if `unlockDelaySec` is too big we will always get 0 withdrawable amount because of

```
if (deltaTime < unlockDelaySec) {
    return 0;
}
```

3. Also in `_getWithdrawableAmount` if `unlockPeriodSec` is too big we will always get 0 because this `uint256 deltaTimeNormalized = (deltaTimeDelayed * precision) / unlockPeriodSec;` will be zero

This means the user will need to wait a huge amount (might be infinite) of time to be able to unlock his funds, and they won't be unlockable even with the `liquidateFunds` functionality

Impact

This can possibly lead to user funds being stuck in Zerem, but this requires misconfiguration in deployment, so it is Medium severity

Recommendation

Add sensible constraints for the valid values of `unlockDelaySec` and `unlockPeriodSec` in the constructor of `Zerem.sol`

Client response

Resolved by adding constraints in the constructor

[M-06] ERC20 tokens that have a fee-on-transfer mechanism require special handling

Proof of Concept

Some tokens take a transfer fee (`STA`, `PAXG`) and there are some that currently do not but might do so in the future (`USDT`, `USDC`). Since Zerem might be integrated with a protocol that works with all types of ERC20 tokens, and Zerem should too, this can lead to problems.

Let's look at the following scenario:

1. Alice tries to claim tokens that have a fee-on-transfer mechanism from a protocol that is integrated with Zerem
2. The integrated protocol calls `Zerem::transferTo` method but the `amount` argument passed does not take the fee into consideration
3. The `require(transferredAmount >= amount, "not enough tokens");` check will always fail, since the `transferredAmount` will be less than `amount` due to the fee

If this happens this means that all of users balances of such tokens won't be claimable and stuck forever.

Impact

If a token with a fee-on-transfer mechanism is used and not properly handled on both the integration protocol and Zerem's side, it can result in 100% stuck balances of this token of users. Since this happens only with a special type of ERC20 it is Medium severity.

Recommendation

Integration of such tokens will require special handling on the integrating protocol side (pre-calculating the fee, so the `amount` argument passed has the correct value) and possibly on Zerem's side. Consider either better documentation for those or advise integrating protocols to not transfer such tokens through Zerem.

Client response

Added a warning comment in the code

[M-07] Protocol does not work with ERC20 tokens that have a mechanism for balance modifications outside of transfers

Proof of Concept

Some tokens may make arbitrary balance modifications outside of transfers. One example are Ampleforth-style rebasing tokens and there are other tokens with airdrop or mint/burn mechanisms. The Zerem system caches the locked balances for users and if such an arbitrary modification has happened this can mean that the protocol is operating with outdated information. Let's look at the following scenario:

1. Alice claims some tokens with such a mechanism from a protocol that is integrated with Zerem
2. The protocol calls `Zerem::transferTo` method, but the amount sent is ≥ `lockThreshold` so the funds are locked
3. In `_lockFunds` the amount sent is cached in `record.totalAmount`, `record.remainingAmount` and `pendingTotalBalances[user]`.
4. Some time after this, let's say a rebase of the token balances has happened and now actually Zerem holds less tokens
5. Now when the `unlockPeriodSec` passes and Alice wants to claim her tokens she is unable to because the cached amount is more than the actual amount that is held in the Zerem protocol, so the transaction always reverts leading to all of the locked funds getting stuck

Also if the rebasing of the tokens actually increased the protocol balance, then those excess tokens will be stuck in it.

Impact

Funds can be stuck in Zerem, but it requires a special type of ERC20 token, so it is Medium severity.

Recommendation

Allow partial unlock of funds or document that the protocol does not support such tokens, so integrating protocols do not transfer them through Zerem. Also you can add functionality to rescue excess funds out of the Zerem protocol.

Client response

Acknowledged

8.3. QA Findings

[QA-01] Use latest Solidity version with a stable pragma statement

Using a floating pragma `^0.8.13` statement is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode. Also use latest Solidity version to get all compiler features, bugfixes and optimizations

[QA-02] Add NatSpec documentation

NatSpec documentation to all public methods and variables is essential for better understanding of the code by developers and auditors and is strongly recommended.

[QA-03] Code is not formatted properly

Use a code formatter to keep code clean & tidy, I'd suggest adding the `forge fmt` command to your pre-commit hook

[QA-04] Move event emissions to the methods where the action happens

Move `TransferFulfilled` event emission to `_sendFunds()` and `TransferLocked` event emission to `_lockFunds()`

[QA-05] Typos in comments

Change `recive` to `receive` and `timeframe` to `time frame`

[QA-06] Missing non-zero address checks in the constructor

Add non-zero address checks for all `address` type arguments in `Zerem.sol`'s constructor

[QA-07] No need to cast to `address payable` in `_sendFunds`

Casting `address` to `address payable` is only needed when using `send` or `transfer`, so it is not needed here

[QA-08] Some functions can be fused into one

Merge `freezeFunds` and `unfreezeFunds` into `updateFundsFreezeStatus` - same for their events. There is no need for two separate methods and events

[QA-09] Open `TODO` in the code

There is an open `TODO` in `unlockFor` - this implies changes that might not be audited. Resolve it or remove it

[QA-10] Remove duplicated code

Reuse code, make `_getRecord` call `_getTransferId` because the `bytes32 transferId = keccak256(abi.encode(user, lockTimestamp));` logic is duplicated, same for `_unlockFor`

[QA-11] Update external dependency to latest version

The OpenZeppelin library dependency is using a stale version - upgrade to latest one to get all security patches, features and gas optimisations

[QA-12] Consider using a differet key in the `pendingTransfers` mapping

Currently the key in the `pendingTransfers` mapping is calculated by this

```
bytes32 transferId = keccak256(abi.encode(user, lockTimestamp));
```

I'd say that just using a simple uint256 nonce for each `TransferRecord` would work just fine. It will also be simpler and more gas efficient.