Smoothly Security Review

Pashov Audit Group

Conducted by: pashov

August 31st, 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 **Smoothly** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Smoothly

The protocol is a "MEV Smoothing Pool" which allows for Ethereum validators to pool together their tips & MEV from block proposals. The idea is that the size of tips & MEV from proposed blocks varies a lot, mostly based on network activity. If individual validators pool their tips together it is more likely they will receive more rewards overall, especially in a short (<10 years) timeframe.

How it works is that registered validator deposits 0.65 ETH (for slashing if they act maliciously) and then he sets the fee recipient address in his proposed blocks to the `SmoothlyPool` contract. Then a group of operators control the protocol governance, deciding how to split rewards to or slash the validators, by submitting votes on-chain with merkle roots.

Observations

Events are a core component of the protocol, as they are used off-chain for critical operations. This allows for high centralization as many things happen off-chain and are not controlled by the Ethereum network and smart contracts.

Operators & Validators should only be added/registered at the beginning of new epochs, so that they are fairly eligible for the rewards in epochs.

Privileged Roles & Actors

- Smoothly Pool owner - can set the `withdrawals`, `exists` and `state` Merkle tree roots as well as withdraw a `fee` amount from the pool, the `PoolGovernance` contract will hold this role
- Pool Governance owner - can manage pool operators and transfer pool ownership
- Pool Governance operator - can propose new `withdrawals`, `exits` and `state` Merkle tree roots as well as a `fee` to be distributed amongst all operators
- Registered validator - can claim pool rewards and/or exit the pool

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 **- 400acaff945f8e41a0ede64e711deb69438d5a2c**

fixes review commit hash **- 028ba5e7eda683f60efd7867d18d1a32f6f16376**

Scope

The following smart contracts were in scope of the audit:

- `PoolGovernance`
- `SmoothlyPool`

7. Executive Summary

Over the course of the security review, pashov engaged with Smoothly to review Smoothly. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	Smoothly
Date	August 31st, 2023

Findings Count

Severity	**Amount**
Critical	2
High	1
Medium	2
Low	5
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Operators can cast an extra vote to get voting majority	Critical	Resolved
[C-02]	Operator can still claim rewards after being removed from governance	Critical	Resolved
[H-01]	Large centralization attack surface	High	Resolved
[M-01]	Operator might be unable to withdraw rewards due to gas limit	Medium	Resolved
[M-02]	The stake fees are not tracked on chain	Medium	Resolved
[L-01]	Disallow epoch proposals when operators.length == 1	Low	Resolved
[L-02]	Small ETH dust amount will be left in PoolGovernance	Low	Resolved
[L-03]	The epoch should be a part of the leafs in the withdrawals & exits Merkle trees	Low	Resolved
[L-04]	Methods are not following the Checks-Effects-Interactions pattern	Low	Resolved
[L-05]	State-changing methods are missing event emissions	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Operators can cast an extra vote to get voting majority

Severity

Impact: High, as it breaks the `votingRatio` invariant

Likelihood: High, as operators can cast an extra vote at any time

Description

In `PoolGovernance::proposeEpoch`, operators can cast an extra vote when this is needed to get to `votingRatio`. Exploiting this, a single operator in a group of three can execute any proposal he decides on. Also if there are more operators in the group and one extra vote is needed for a proposal, anyone who has already voted can execute the proposal by sending his vote again. This is due to how the mechanism for removing previous votes works:

```
bytes32 prevVote = votes[epochNumber][msg.sender];
uint256 count = ++voteCounter[epochNumber][vote];
uint256 operatorsLen = operators.length;

votes[epochNumber][msg.sender] = vote;

if (prevVote != bytes32(0)) --voteCounter[epochNumber][prevVote];
```

The `count` is updated without checking if the user has already voted, meaning if he already voted and casted the same vote, the `count` value will be 2, instead of 1.

A single operator can directly execute a proposal when:

- `operators.length == 1` - can directly execute
- `operators.length == 2` - can directly execute
- `operators.length == 3` - can directly execute
- `operators.length == 4` - now if other operators have voted the single operator can double vote for any of their votes and directly execute it
- `operators.length == 5` - same as `operators.length == 4`

Bigger `operators.length` means that a single operator can't execute a proposal by himself, but using a double vote is always possible.

Add this test to `PooGovernance.t.ts` to run the Proof of Concept

```
it("Operators can cast an extra vote to get voting majority", async () => {
  await governance.addOperators([
    operator1.address,
    operator2.address,
    operator3.address,
  ]);
  await time.increase(week);
  await governance
    .connect(operator1)
    .proposeEpoch([withdrawals.root, exits.root, state, fee]);

  expect(await governance.epochNumber()).to.equal(0);
  // operator1 casts a second vote to get 66% vote ratio
  await governance
    .connect(operator1)
    .proposeEpoch([withdrawals.root, exits.root, state, fee]);

  // validate that the epoch increased (vote passed)
  expect(await governance.epochNumber()).to.equal(1);
});
```

Recommendations

Change the code in the following way:

```
bytes32 prevVote = votes[epochNumber][msg.sender];
- uint256 count = ++voteCounter[epochNumber][vote];
uint256 operatorsLen = operators.length;

votes[epochNumber][msg.sender] = vote;

if (prevVote != bytes32(0)) --voteCounter[epochNumber][prevVote];
+ uint256 count = ++voteCounter[epochNumber][vote];
```

[C-02] Operator can still claim rewards after being removed from governance

Severity

Impact: High, as rewards shouldn't be claimable for operators that were removed from governance

Likelihood: High, as this will happen every time this functionality is used and an operator has unclaimed rewards

Description

The `deleteOperators` method removes an operator account from the `PoolGovernance` but it still leaves the `operatorRewards` mapping untouched, meaning even if an operator is acting maliciously and is removed he can still claim his accrued rewards. This shouldn't be the case, as this functionality is used when operators must be slashed. Also if an operator becomes inactive, even if he is removed, his unclaimed rewards will be stuck in the contract with the current implementation.

Recommendations

On operator removal transfer the operator rewards to a chosen account, for example the `SmoothlyPool`.

8.2. High Findings

[H-01] Large centralization attack surface

Severity

Impact: High, as multiple authorized actors can act maliciously to steal funds

Likelihood: Medium, as it requires malicious or compromised actors, but it's not just the protocol `owner`

Description

1. The `owner` of `PoolGovernance` can add a big number of operators, resulting in a block gas limit DoS of `proposeEpoch`, because the method loops over all operators
2. Operators can group their voting and claim all of `SmoothlyPool`'s balance as a `fee`
3. Operators can allow an unregistered account to claim rewards from `SmoothlyPool` by adding it to one of the `withdrawals` Merkle tree leafs
4. The `owner` of `PoolGovernance` can transfer ownership of `SmoothlyPool` to an account he controls and directly call `updateEpoch`, withdrawing the whole contract balance as the `fee`

Recommendations

Make the `owner` of `PoolGovernance` be a multi-sig wallet behind a Timelock contract so that users can monitor what transactions are about to be executed by this account and take action if necessary. Also add a limit on the max number of operators, for example 50. For the operators you might need to add some extra security mechanism to protect the centralization, as currently it doesn't have an easy fix.

8.3. Medium Findings

[M-01] Operator might be unable to withdraw rewards due to gas limit

Severity

Impact: High, as operator's yield will be frozen

Likelihood: Low, as it requires operator to be a special multi-sig wallet or contract

Description

The `PoolGovernance:withdrawRewards` method allows operators to withdraw their yield, which happens with this external call:

```
(bool sent, ) = msg.sender.call{value: rewards, gas: 2300}("");
```

The 2300 gas limit might not be enough for smart contract wallets that have a `receive` or `fallback` function that takes more than 2300 gas units, which is too low (you can't do much more than emit an event). If that is the case, the operator won't be able to claim his rewards and they will be stuck in the contract forever.

Recommendations

Remove the gas limit from the external call. It can also be removed from the same logic in `SmoothlyPool` as well.

[M-02] The stake fees are not tracked on chain

Severity

Impact: High, as it can result in wrong accounting of ETH held by `SmoothlyPool`

Likelihood: Low, as it requires off-chain code to be wrong

Description

Every validator who joins the `SmoothlyPool` should register by paying a `STAKE_FEE` (with the size of 0.065 ETH) to the contract. The pool does not track how much of a stake fee balance a validator has, which is problematic for the following reasons:

1. The pool has no guarantee that it holds at least `numValidators * STAKE_FEE` ETH in its balance - the ETH might have been mistakenly distributed as rewards or claimed as fees from operators
2. It is possible for a validator to deposit more than `STAKE_FEE` if he calls `SmoothlyPool::addStake` multiple times
3. The slashing/punishment mechanism can't be enforced on chain

Recommendations

Add a mapping to track validators' stake fee balances in `SmoothlyPool`.

8.4. Low Findings

[L-01] Disallow epoch proposals when `operators.length == 1`

When `operators.length == 1` the operator will be able to execute any epoch proposal, which shouldn't be allowed. In `proposeEpoch` check that `operators.length` is at least 2 to prevent this scenario.

[L-02] Small ETH dust amount will be left in `PoolGovernance`

In `PoolGovernance::proposeEpoch` the math for calculating how much each operator gets from the `epoch.fee` is `uint256 operatorShare = epoch.fee / operatorsLen;`. Because of how Solidity and EVM executes divisions, it is quite possible to have some remainder after the division, which will be the ETH amount that is left (multiplied by the number of operators) in the `PoolGovernance` contract after all operators withdraw their rewards. Consider a mechanism to distribute leftover dust.

[L-03] The `epoch` should be a part of the leafs in the `withdrawals` & `exits` Merkle trees

In the case of a root reuse, it is safe to have the `epoch` number as part of the leafs in the Merkle trees, so previous epoch's leafs can't be reused. Add this to both `withdrawRewards` and `withdrawStake` in `SmoothlyPool`

[L-04] Methods are not following the Checks-Effects-Interactions pattern

The `PoolGovernance::proposeEpoch` and `SmoothlyPool::updateEpoch` methods are not following the CEI pattern. Make sure that you do external calls last in those methods.

[L-05] State-changing methods are missing event emissions

While the `SmoothlyPool` contract is highly dependent on events, the `PoolGovernance` does not emit events on state changing methods. Also, `SmoothlyPool::receive` doesn't emit one. Make sure that all methods in the codebase that change state emit proper events for off-chain monitoring.