SOLIDIFIED

Audit Report for SKALE SRW - January 26, 2023

Summary

Audit Report prepared by Solidified covering the SKALE self-recharging wallet (SRW) contract.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on January 13, 2023, and the results are presented here.

Audited Files

The source code has been supplied in the following source code repository:

Repo: https://github.com/skalenetwork/skale-manager

The scope consisted of the file contracts/Wallet.sol at commit 43ba95d1d049b57332339ca548e6784a2e68ce75

Intended Behavior

The audited contracts implement self-recharging wallets that can be replenished by anyone and are used to reimburse nodes.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	High	-

Issues Found

Solidified found that the SKALE SRW contracts contain no critical issues, no major issues, 2 minor issues, and 3 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	transfer used for sending ETH	Minor	Acknowledged
2	Wallets.sol: Owner of Wallets.sol can call withdrawFundsFromSchainWallet() even if the Schain is not deleted	Minor	Fixed
3	Gas calculation logic in refundGasBySchain can be wrong when multiple calls happen in a transaction	Note	Acknowledged
4	withdrawFundsFromValidatorWallet has an unnecessary require statement	Note	Acknowledged
5	Wallets.sol: No Storage Gap in Upgradeable Contract	Note	Acknowledged

Critical Issues

No critical issues have been found.

Major Issues

No major issues have been found.

Minor Issues

1. `transfer` used for sending ETH

The contract uses `transfer` in multiple places for sending ETH.

As `transfer` only comes with a 2300 gas stipend, its use is discouraged. If the owner is for instance a multi-sig wallet with some complicated business logic in its `receive` function, the transaction will revert.

Recommendation
Use `call` for sending ETH.

Status
Acknowledged.

2. `Wallets.sol`: Owner of Wallets.sol can call `withdrawFundsFromSchainWallet()` even if the Schain is not deleted

In the documentation for `withdrawFundsFromSchainWallet()`, it is mentioned as a requirement that it should only be called after initializing the deletion of an Schain, which is true if the method is called via the `Schains` contract.

However, the owner of `Wallets` can call this method at any point in time, withdraw funds from the Schain, and transfer it to the Schain owner.

Without any funds, the Schain would stop working even when it is not supposed to be stopped.

Recommendation

Add a check using `SchainsInternal` to check whether the Schain is active or not by using `schainsInternal.isSchainActive(schainHash)`, and only allow the owner of `Wallets` to withdraw to the owner of Schain when the Schain is inactive.

Status

Fixed.

Informational Notes

3. Gas calculation logic in `refundGasBySchain` can be wrong when multiple calls happen in a transaction

If `isDebt` is true, 21,000 or 6,000 gas is added to the amount (depending on if the debt value is previously 0) to account for the additional `SSTORE`. While these values are sensible when this is the first operation within a transaction that modifies `_schainDebts`, they can be wrong if this is not the case. As we can see in the `SSTORE` section within evm.codes, it does not only matter if the value was previously 0, but the gas cost also depends on the value of the slot before the transaction. Therefore, if `_schainDebts` was non-zero before the transaction, is set to 0 within the transaction (in `refundGasByValidatorToSchain`) and then set to a non-zero value in `refundGasBySchain` again, the calculation will be wrong and add too much gas.

Recommendation
Measure the actual gas cost. This will require an additional `SSTORE`, but this store will always be on a non-zero warm slot, resulting in negligible additional cost.

Status
Acknowledged: The gas calculation is only an approximation.

4. `withdrawFundsFromValidatorWallet` has an unnecessary require statement

The statement `require(amount <= _validatorWallets[validatorId], "Balance is too low");` can be removed to save gas since the subtraction below will result in an underflow and revert.

Recommendation

Remove the `require` statement.

Status

Acknowledged.

5. `Wallets.sol`: No Storage Gap in Upgradeable Contract

`Wallets.sol` inherits from `Permissions.sol`, which is an upgradeable contract, but it is missing a storage gap. Inheriting contracts may introduce new variables.

In order to be able to add new variables to the upgradeable abstract contract without causing storage collisions, a storage gap should be added to the upgradeable contract.
If no storage gap is added, when the upgradable contract introduces new variables, it may override the storage slots of variables in the inheriting contract.

Recommendation

Consider adding a storage gap: `uint256[50] private __gap`

Status

Acknowledged.

Disclaimer