byterocket

Smart Contract Audit Report

Sovreign

5th of August 2021

Contents

1. Preface

The team of **Sovreign** contracted byterocket to conduct a smart contract audit of their smart contracts. Sovreign is "*a decentralized finance protocol with store-of-value assets reigning supreme for the global trade market*".

The team of byterocket reviewed and audited their smart contracts in the course of this audit. We started on the 14th of June and finished on the 5th of August.

The audit included the following services:

- *Manual Multi-Pass Code Review*
- *Automated Code Review*
- *In-Depth Protocol Analysis*
- *Deploy & Test on our Testnet*
- *Formal Report*

byterocket gained access to the code via their private GitHub repository. We started the audit on the master branch's state on June 11st, 2021 (*commit hash a54990e712bfc00c5281dce9aed1b9edbde7f5c3*).
Throughout our audit, changes have been made to the code, which we incorporated up until the latest state of the master branch on July 28th, 2021 (*commit hash 01412cac872b5d105cf52268f0e55fbe40cb500b*).

2. Manual Code Review

We conducted a manual multi-pass code review of the smart contracts mentioned in section (1). Three different auditors went through the smart contract independently and compared their results in multiple concluding discussions.

These contracts are written according to the latest standards used within the Ethereum community and the Solidity community's best practices. The naming of variables is very logical and understandable, which results in the contract being easy to understand. The code is very well documented and up to the latest standards. We were very satisfied with the overall quality of the code.

On the code level, we **found one bug or flaw of low severity**. Besides that, we have noted some very minor other findings. A further check with multiple automated reviewing tools (*MythX*, *Slither*, *Manticore*, *and different fuzzing tools*) **did not find any additional bugs** besides some common false positives.

2.1 Bugs & Vulnerabilities

2.1.A - VestingRouter.sol - Line 33 [LOW SEVERITY]

```
if (
  _reign.balanceOf(address(this)) < _vestingAmount[i] ||
  gasleft() < 20000
) {
  break;
}
```

The logic in line 33 (`gasleft() < 20000`) prevents the loop from executing with too little gas left for the execution to successfully complete in order to prevent reverts. However, as per our knowledge, even the cheapest ERC20 transfers still use 34.000 gas units on average. Even when being executed in batches, very cheap ERC20 implementations never use less gas than roughly 24.000 gas units.
In this case, the call would revert if the first condition in line 32 is false while the leftover gas units are greater than 20.000 but smaller than the required amount.

We suggest that the Sovreign team exactly finds out how much gas is being used in the worst case scenario and to adapt the value in line 33 to it.

2.2 Other Findings

2.2.A - ReignFacet.sol - Line 18 - 20 [NO SEVERITY]

```
mapping(uint128 => bool) isInitialized;
mapping(uint128 => uint256) initialisedAt;
mapping(address => uint256) balances;
```

These mappings don't have a visibility defined. This does not pose any risks as the default visibility for variables is `internal`, however, we always advise developers to explicitly set the visibility, even if it's just for the sake of readability.

2.2.B - ReignFacet.sol - Line 231 + 242 [NO SEVERITY]

```
// balanceAtTs returns the amount of BOND that the user currently staked
(bonus NOT included)
```

The two comments in line 231 and 242 are referencing the BOND token, which should probably be REIGN instead, as there is no BOND token issued by these smart contracts.

2.2.C - LPRewards.sol - Line 29 [NO SEVERITY]

```
address private _rewarewardsVault;
```

This is probably a typo of `_rewardsVault`.

2.2.D - VestingRouter.sol - Line 10 + 11 [NO SEVERITY]

```
address[] public _vestingAddress;
uint256[] public _vestingAmount;
```

Throughout the code base, array variable names are referred to in their plural forms, while it is not done here. We suggest using `_vestingAddresses` as well as `_vestingAmounts`.

2.2.E - Bridge.sol - Line 39 [NO SEVERITY]

```
uint256 eta
```

The time lock aspect of the DAO is enforced in the ReignDAO contract itself, while the `eta` variable is not being used in the Bridge contract itself anymore (in contrast to the original implementation by BarnBridge). The variable serves no real purpose in the Bridge contract anymore, thus we would probably remove it, but as it makes no real difference we leave it up to the Sovreign developers to decide this. We are just noting it for the sake of completeness.

2.2.F - ReignDAO.sol - Line 92-94 [NO SEVERITY]

```
IReign reign;
IBasketBalancer basketBalancer;
bool isInitialized;
```

These variables don’t have a visibility defined. This does not pose any risks as the default visibility for variables is `internal`, however, we always advise developers to explicitly set the visibility, even if it’s just for the sake of readability.

3. Protocol/Logic Review

Part of our audits are also analyses of the protocol and its logic. A team of three auditors went through the implementation and documentation of the implemented protocol.

We were **not able to find any weaknesses or underlying problems** of a game theoretic nature in the protocol and it's logic. We were also not able to think of any loopholes that were not covered by the Sovreign developers.

The code base itself is quite complex and consists of a multitude of different contracts. The developers did a good job of separating the logic into the different contracts and contract groups.

Sovreign is making use of the **EIP-2535 Diamond Cut standard**, which provides a lot of benefits with the downside of an increased complexity level in comparison to "*old-fashioned*" proxy upgradeable approaches. Their implementation of the Diamond Cut smart contract architecture adheres to the standard and best practices used within the group of projects that make use of this architecture. We have not found any issues with this implementation and its logic.

In our internal discussion we were not able to find any weaknesses within the implementation of the Sovreign protocol itself. We have extensively read and studied the whitepaper (*which can be found on their website*) and discussed it internally given the provided implementation. We came to the conclusion that the described protocol has correctly been implemented. There are most certainly other ways to reach the same target, ranging from the choice of the Diamond Cut architecture to the usage of Balancer-based pools, but we agree that the chosen technologies and architectures are most likely the best choice given the team's requirements.

On the code level, none of our function calls (even when we did randomized fuzzing tests) were able to produce any unforeseen outcomes and skew the results in any way. The overall process flow was valid at all times.

A lot of problems and issues usually occur when forked projects are modified to meet developer requirements. The Sovreign developers have shown to deeply understand the code that they have used as inspiration for their project (*mainly BarnBridge*). They have managed to make their changes to them according to a great overall design and most importantly without creating security vulnerabilities.

We were **not able to discover any problems** in the protocol implemented in the smart contract.

4. Testnet Deployment

As per our testing strategy, we deploy audited smart contracts (*if requested by the client*) onto a testnet to verify the implementation's functionality. We usually deploy simple smart contracts to a local Ganache instance, which is sufficient for most cases. In this case. we wanted to ensure no Ganache-related coverups of the contracts' misbehavior. We created two testnets: a geth-based one and a parity/openethereum-based one. All of our tests have been **executed on both testnets without any difference in the results**. We were able to use the contracts as intended and could not maliciously game the protocol in practice.

We used fuzzing tools that generate random and semi-random inputs and interact with the contracts, trying to produce any unforeseen states within the contracts, especially the treasury contract. Throughout our tests, we were **not able to create or observe any issues or problems**. The contract behaved as expected and reverted correctly, given wrong inputs. No unforeseen states occurred during our fuzz-tests.

Since the overall smart contract architecture is quite complex, we also tested the individual contracts specifically for potential problems regarding the access logic like onlyOwner, onlyDAO or onlyRouter. Throughout these tests, we were not able to interact with any contract function without the proper access rights.

5. Unit Tests and Coverage

When looking at how the contracts behave in practice, we also take a look at the provided unit tests. The overall quality of the repository and its structure is very high, which does also include well-written unit tests with excellent coverage.

All of the current best-practice checks have been implemented. We have not found any immediate need for improvement on the current tests that would have led us to believe that certain areas of the code are un- or under-tested. We were also very pleased with the excellent coverage. During our tests, the coverage of the provided tests was mostly hovering around 100% for the majority of contracts and their functions/lines, while the rest was in the 95+% area.

In our tests none of the provided unit tests failed. All in all, we are very satisfied with the provided unit tests, their quality and the overall test coverage. We were **not able to find any issues** with the provided tests.

6. Summary

During our code review (*which was done manually and automated*), **we found one bug or flaw of low severity**. Besides that, we have noted some very minor other findings. A further check with multiple automated reviewing tools **did not find any additional bugs** besides some common false positives.

The protocol review and analysis did neither uncover any game-theoretical nature problems nor any other functions prone to abuse. According to our analysis, the Sovreign protocol has been correctly implemented in the smart contracts as the whitepaper describes it. We were **not able to find any issues**.

During our multiple deployments to various local testnets, we haven't been able to find any additional problems or unforeseen issues. The provided unit tests worked flawlessly and we were also **not able to find any issues** here. Throughout our tests, their reported coverage was excellent.

In general, we are **delighted** with the overall quality of the code, its tests and documentation.