Report

v. 3.0

Customer

RedStone

Smart Contract Audit

Token and Vesting

31st March 2023

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	28.03.23	A. Zveryanskaya	Initial Draft
0.2	28.03.23	A. Zveryanskaya	Minor revision
1.0	28.03.23	A. Zveryanskaya	Release
1.1	31.03.23	A. Zveryanskaya	CVF-3, 4 are downgraded
2.0	31.03.23	A. Zveryanskaya	Release
2.1	31.03.23	A. Zveryanskaya	CVF-3, 4 Client comments are updated
3.0	31.03.23	A. Zveryanskaya	Release

2 Introduction

The following document provides the result of the audit performed by ABDK Consulting (Mikhail Vladimirov and Dmitry Khovratovich) at the customer request. The audit goal is a general review of the smart contracts structure, critical/major bugs detection and issuing the general recommendations.

RedStone builds a novel oracle that delivers data feeds for 1,100+ assets, available on Ethereum, Polygon, zkEVM, Avalanche, Arbitrum, and 30+ chains with 10 second update time. Thanks to RedStone's StorageLess architecture, which bypasses expensive EVM-storage, projects can integrate it once and use on all EVM chains.

- https://docs.redstone.finance/
- https://app.redstone.finance/

3 Project scope

We were asked to review:

- Original Code
- Code with Fixes

Files:

```
/
        LockingRegistry.sol     RedstoneToken.sol     VestingWallet.sol
```

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check if code actually does what it is supposed to do, if that algorithms are optimal and correct, and if proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Minor issues** contain code style, best practices and other recommendations.

5 Our findings

We found 2 critical, 1 major, and a few less important issues. All identified Critical and Major issues have been fixed.

Issues	Active	Fixed
Severity: Critical	0	2
Major	0	1
Moderate	4	1
Minor	0	15

Fixed 19 out of 23 issues

6 Critical Issues

CVF-1. FIXED

- **Category** Flaw
- **Source** VestingWallet.sol

Description This function could be called by anyone. It effectively overwrites the previous unlock request and resets the unlock timer. Thus, a malicious actor could use this function to effectively prevent the beneficiary from ever unlocking his tokens.

Client Comment *Added onlyBeneficiary modifier to the VestingWallet contract.*

```
94 function requestUnlock(uint256 amount) external {
```

CVF-2. FIXED

- **Category** Flaw
- **Source** VestingWallet.sol

Description This function could be called by anyone. A malicious actor could use it to prevent the beneficiary from releasing his tokens.

Client Comment *Added onlyBeneficiary modifier to the VestingWallet contract.*

```
85 function lock(uint256 amount) external {
```

7 Major Issues

CVF-5. FIXED

- **Category** Suboptimal
- **Source** VestingWallet.sol

Description The expression “getUnvestedAmount(block.timestamp)” is calculated twice.

Recommendation Consider calculating once and reusing.

```
67  token.balanceOf(address(this)) < getUnvestedAmount(block.timestamp)

69    : token.balanceOf(address(this)) - getUnvestedAmount(block.
        ↪ timestamp);
```

8 Moderate Issues

CVF-3. INFO

- **Category** Overflow/Underflow
- **Source** VestingWallet.sol

Description Phantom overflow is possible here.

Recommendation Consider using the “muldiv” function.

Client Comment *IMO the severity is too high. The muldiv function looks like an unneccessary overcomplication, I didn’t add SafeMath as well because it is integrated by default to Soidity 0.8.x.*

(start + cliffDuration + vestingDuration − timestamp)
can be safely limited by
10_000 years = 10000 ∗ 365 ∗ 24 ∗ 3600 seconds = 315360000000

The ‘allocation‘ can be limited to
total_supply ∗ precision
(currently 50_000_000e18 but shouldn’t increase more than to 1_000_000_000e18 1e27),but to cause the phantom overflow it should be greater than
2^ 256 / 315360000000 = 3.6717431e+65 *which will never be the case.*

```
58 (allocation * (start + cliffDuration + vestingDuration - timestamp))
     ↪  / vestingDuration;
```

CVF-4. INFO

- **Category** Unclear behavior
- **Source** VestingWallet.sol

Description This calculation rounds down, i.e. towards user. A best practice is to always round towards the protocol.

Recommendation Consider rounding up.

Client Comment *IMO the severity is too high. The only thing that may happen because of the rounding towards user (not protocol) is that the funds (extremely small fraction of them: max 1 wei) can become withdrawable a bit sooner (max 1 second sooner comparing to the approach with rounding towards protocol).*
It will not allow beneficiary to drain contract funds and will not cause withdrawals locking, because in the end of vesting period the beneficiary will still be able to withdraw all tokens from the contract:

withdrawableAmountAfterVestingEnd=token.balanceOf(address(this))−unvestedAmount

(unvestedAmount will be equal to 0)

```
58 (allocation * (start + cliffDuration + vestingDuration - timestamp))
     ↪  / vestingDuration;
```

CVF-6. INFO

- **Category** Unclear behavior
- **Source** LockingRegistry.sol

Description In case the locked amount is not enough, the user is not allowed to withdraw anything.

Recommendation Consider allowing to withdraw the locked amount in such a case.

Client Comment *It is intended. This case indicates that some funds were slashed after the unlock request. Which means that this data provider probably violated the rules and tried to withdraw their stake. It will give us a bit more time to analyze its behaviour thanks to the additional waiting time.*

```
70 require(amountToUnlock <= userLockingDetails.lockedAmount, "Can not
      ↪ unlock more than locked");
```

CVF-7. INFO

- **Category** Unclear behavior
- **Source** LockingRegistry.sol

Description In case the locked amount is less than the slashed amount, this check doesn't allow to slash anything.

Recommendation Consider allowing to slash the locked amount in such a case.

Client Comment *IMO it would cause unclear behaviour of this function. It would be misleading that the transaction succeeded with the given 'slashedAmount' argument, effectively slashing less.*

```
90 require(
     userLockingDetails.lockedAmount >= slashedAmount,
     "Locked balance is lower than the requested slashed amount"
   );
```

CVF-8. FIXED

- **Category** Unclear behavior
- **Source** VestingWallet.sol

Description The returned value is ignored.

Recommendation Consider explicitly checking that true was returned.

```
90 token.approve(address(lockingRegistry), amount);
```

9 Minor Issues

CVF-9. FIXED

- **Category** Procedural
- **Source** LockingRegistry.sol

Recommendation Consider specifying as “^0.8.0” unless there is something special with this particular version.

```
3 pragma solidity ^0.8.4;
```

CVF-10. FIXED

- **Category** Bad naming
- **Source** LockingRegistry.sol

Recommendation Events are usually named via nouns, such as “UnlockRequest” and “Unlock”.

```
19 event UnlockRequested(address user, UserLockingDetails
   ↪ lockingDetails);
20 event UnlockCompleted(address user, UserLockingDetails
   ↪ lockingDetails);
```

CVF-11. FIXED

- **Category** Suboptimal
- **Source** LockingRegistry.sol

Recommendation The “user” parameters should be indexed.

```
19 event UnlockRequested(address user, UserLockingDetails
   ↪ lockingDetails);
20 event UnlockCompleted(address user, UserLockingDetails
   ↪ lockingDetails);
```

CVF-12. FIXED

- **Category** Bad datatype
- **Source** LockingRegistry.sol

Recommendation The type of this argument should be “IERC20”.

```
28 address lockedTokenAddress,
```

CVF-13. FIXED

- **Category** Procedural
- **Source** LockingRegistry.sol

Recommendation This check should be performed earlier.

```
46 require(amountToUnlock > 0, "Amount to unlock must be a positive
   ↪ number");
```

CVF-14. FIXED

- **Category** Procedural
- **Source** LockingRegistry.sol

Recommendation This check should be performed earlier.

```
64 require(
     block.timestamp > userLockingDetails.unlockOpeningTimestampSeconds
       ↪ ,
     "Unlocking is not opened yet"
   );
```

CVF-15. FIXED

- **Category** Procedural
- **Source** RedstoneToken.sol

Recommendation Consider specifying as “^0.8.0” unless there is something special with this particular version.

```
3 pragma solidity ^0.8.4;
```

CVF-16. FIXED

- **Category** Procedural
- **Source** RedstoneToken.sol

Description Defining top-level constants in a file named after a contract makes it harder to navigate through code.

Recommendation Consider either moving the constants into the contract or moving them into a separate file.

```
7 uint constant MAX_SUPPLY = 50000000 * 1e18;
```

CVF-17. FIXED

- **Category** Suboptimal
- **Source** RedstoneToken.sol

Recommendation This value could be rendered as “50_000_000e18”.

```
7 uint constant MAX_SUPPLY = 50000000 * 1e18;
```

CVF-18. FIXED

- **Category** Unclear behavior
- **Source** RedstoneToken.sol

Description This function should emit some event.

```
27 function updateMinter(address newMinter) external {
```

CVF-19. FIXED

- **Category** Documentation
- **Source** RedstoneToken.sol

Description Typo in the line.

```
28 require(msg.sender == minter, "RedstoneToken: minter update by an
   ↪ unatuthorized address");
```

CVF-20. FIXED

- **Category** Bad naming
- **Source** VestingWallet.sol

Recommendation Events are usually named via nouns, such as “Release”.

```
16 event TokensReleased(uint256 amount);
```

CVF-21. FIXED

- **Category** Bad datatype
- **Source** VestingWallet.sol

Recommendation The type of this variable should be “IERC20”.

```
18 ERC20 public token;
```

CVF-23. FIXED

- **Category** Bad datatype
- **Source** VestingWallet.sol

Recommendation The type of this argument should be “IERC20”.

```
27 address vestingToken_,
```

CVF-24. FIXED

- **Category** Bad datatype
- **Source** VestingWallet.sol

Recommendation The type of this argument should be “LockingRegistry”.

```
29 address lockingRegistry_,
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 40 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting