Zellic

Reserve Throttle Wallet

Smart Contract Security Assessment

August 17, 2023

Prepared for:

The Reserve Protocol team

Reserve

Prepared by:

Sina Pilehchiha and Yuhang Wu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Reserve from August 14th to August 16th, 2023. During this engagement, Zellic reviewed Reserve Throttle Wallet's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there a risk of theft of funds held by the wallet?
- Is there a risk of losing access to funds held by the wallet?
- Is there a risk of losing access-control privileges?
- Is there a possibility of making unpermitted changes to access-control privileges?
- Are there ways to bypass the predefined throttle limits and timelock?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, the unfinished `renounceAdmin` function prevented us from fully verifying its correctness.

1.3 Results

During our assessment on the scoped Reserve Throttle Wallet contracts, we discovered one finding, which was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for

Reserve's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	0

2 Introduction

2.1 About Reserve Throttle Wallet

Reserve Throttle Wallet is a protocol for deploying and managing ERC-20 tokens that are issuable and redeemable for baskets of other tokens.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas

optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Reserve Throttle Wallet Contracts

Repository	https://github.com/reserve-protocol/throttle-wallet
Version	throttle-wallet: `ceba290a0b74a80c3eab147c9c98e5590da2e88d`
Program	ThrottleWallet
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-days. The assessment was conducted over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Sina Pilehchiha, Engineer
sina@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 14, 2023	Kick-off call
August 15, 2023	Start of primary review period
August 16, 2023	End of primary review period

3 Detailed Findings

3.1 The `lastWithdrawalAt` is not initialized in `constructor`

- **Target**: ThrottleWallet
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `lastWithdrawalAt` variable is used in the `availableToWithdraw` function with its default value of zero.

During the first call to the `initiateWithdrawal` function, the `availableToWithdraw` function will be invoked, and at that point, the `lastWithdrawalAt` variable will still have its default value of zero. This may not be the intended behavior and could result in incorrect calculations.

```
function availableToWithdraw() public view returns (uint256) {
       uint256 timeSinceLastWithdrawal = block.timestamp
    - lastWithdrawalAt;
        ...
   }
```

Impact

In the current design, `lastWithdrawalAt` is initially assigned a default value of zero. Consequently, the initial `timeSinceLastWithdrawal` value will effectively be the time from timestamp 0 until the current block timestamp. This could lead to an unexpectedly high value and possibly cause incorrect calculations or inconsistencies with expectations of how the contract should ideally operate.

Recommendations

Initialize the `lastWithdrawalAt` variable in the `constructor` function, setting it equal to the current block's timestamp (`block.timestamp`). This will ensure that the variable does not start at zero.

Remediation

This issue has been acknowledged by Reserve.

Reserve states that this is done intentionally to start the throttle at max instead of zero.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Centralization risks

The ThrottleWallet contract has been designed to hold the treasury of RSR tokens and then allow the `USER` to withdraw them slowly over time as needed. The contract splits roles between an `ADMIN` and a `USER` role and grants each role restrictions and privileges according to the specifications. The contract has some centralization risks, primarily due to the privileges associated with the `ADMIN` role.

This table documents the roles defined in the ThrottleWallet contract and highlights the potential risks associated with them according to the contract and the formal specifications provided by Reserve:

Role	Privileges	Restrictions
`ADMIN`	Set the address of the `USER`, renounce its role as `ADMIN`, and cancel a withdrawal	Arbitrarily set a new `ADMIN` address, initiate a withdrawal, and complete a withdrawal
`USER`	Initiate a withdrawal and complete a withdrawal	Change the address of the `USER`, change the address of the `ADMIN`, and cancel a withdrawal

Please note the `ADMIN` role is able to assign the `USER` role to themselves or an arbitrary third party and then initiate and approve withdrawals without being disturbed by the `onlyUser` modifiers.

Consider implementing a multi-sig mechanism for the `ADMIN` role. Requiring multiple signatures to perform critical actions in the contract significantly reduces the risks associated with one compromised key or the presence of malicious actors. Proper design and implementation of the roles as well as adoption of best key-custody practices can help mitigate the risk and impact of a compromise of a privileged account.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: ThrottleWallet.sol

Function: `cancelWithdrawal(uint256 _nonce)`

This allows cancelling a withdrawal if it has not been completed already.

Inputs

- `_nonce`
  - **Control**: Could be controlled by the admin.
  - **Constraints**: Less than `nextNonce`.
  - **Impact**: Used to specify which withdrawal to cancel.

Branches and code coverage (including function calls)

Intended branches

- Cancel the withdrawal successfully.
  ☑ Test coverage

Negative behavior

- Cancel withdrawal failed and then revert.
  ☑ Negative test

Function: `changeUser(address _newUser)`

This changes the current user to a new one.

Inputs

- `_newUser`

  - **Control**: Could be controlled by the admin.
  - **Constraints**: N/A.
  - **Impact**: New user's address that will be used.

Branches and code coverage (including function calls)

Intended branches

- Change the user to a new one successfully.
  ☑ Test coverage

Negative behavior

- Revert due to the same user before and after.
  ☑ Negative test

Function: `completeWithdrawal(uint256 _nonce)`

This allows completing a withdrawal after the timelock period has passed.

Inputs

- `_nonce`
  - **Control**: Could be controlled by anyone.
  - **Constraints**: Less than `nextNonce`.
  - **Impact**: Used to specify which withdrawal to complete.

Branches and code coverage (including function calls)

Intended branches

- Complete withdrawal successfully.
  ☑ Test coverage
- The balance of each account is right after the withdrawal.
  ☑ Test coverage

Negative behavior

- Complete withdrawal failed and revert.
  ☑ Negative test

Function call analysis

- `completeWithdrawal` → `throttledToken.safeTransfer(withdrawal.target, withdrawal.amount)`

- **What is controllable?** `withdrawal.target` and `withdrawal.amount`.
- **If return value controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `initiateWithdrawal(uint256 amount, address target)`

This initiates withdrawal with a specific amount and target.

Inputs

- `amount`
  - **Control**: Could be controlled by the user.
  - **Constraints**: Should be greater than zero and less than `accumulatedWithdrawalAmount`, and it should not be larger than `balanceOf(address(this)) - totalPending`.
  - **Impact**: The amount to be withdrawn.
- `target`
  - **Control**: Could be controlled by the user.
  - **Constraints**: Should be a valid address.
  - **Impact**: The address to receive the withdrawal.

Branches and code coverage (including function calls)

Intended branches

- Initiate withdrawal successfully.
  ☑ Test coverage

Negative behavior

- Initiate withdrawal failed and revert.
  ☑ Negative test

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Reserve Throttle Wallet contracts, we discovered one finding, which was of low impact. Reserve acknowledged the finding and implemented a fix.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.