Zellic

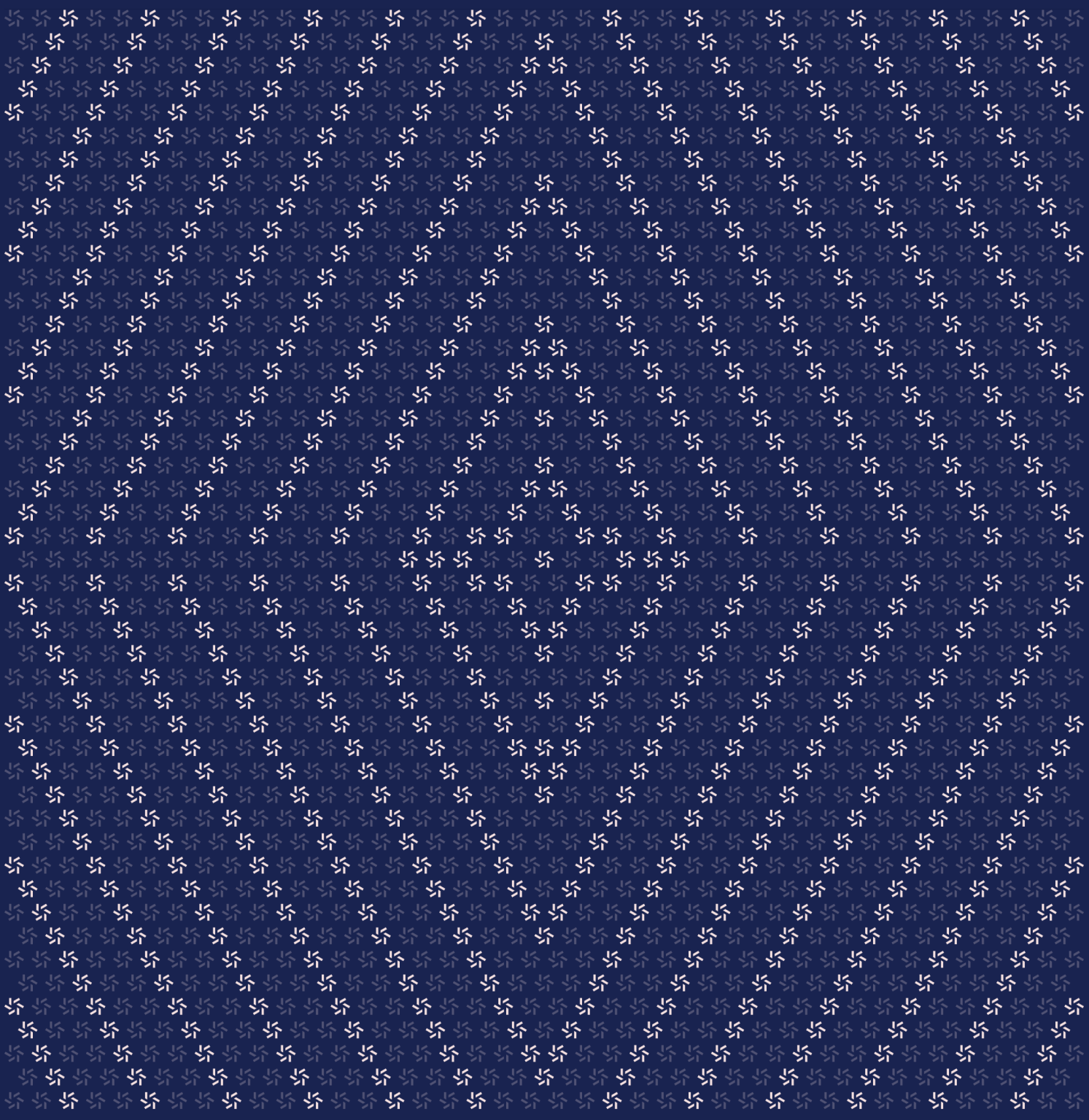
Prepared for
Ming Wu
Blast Futures

Prepared by
Seunghyeon Kim
Jinseo Kim
Zellic

May 8, 2024

RabbitX

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Blast Futures from May 7th to May 8th, 2024. During this engagement, Zellic reviewed RabbitX's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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 the low-level transfer call implemented well?
- Is the EIP-712 verification implemented well?
- Could an on-chain attacker drain the asset?

1.3. 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
- Off-chain projects

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

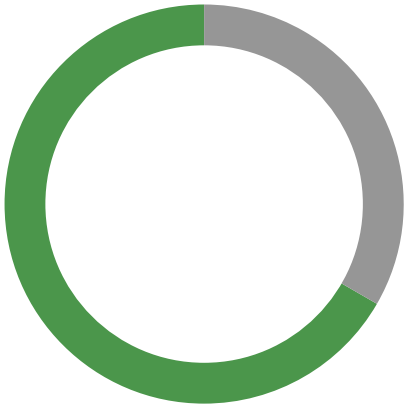
1.4. Results

During our assessment on the scoped RabbitX contracts, we discovered three findings. No critical issues were found. Two findings were of low impact and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Blast Futures's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	1



2. Introduction

2.1. About RabbitX

Blast Futures contributed the following description of RabbitX:

> RabbitX is a global permissionless perpetuals and derivatives exchange.

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 codebase 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, upgradability 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:

RabbitX Contracts

Repository	https://github.com/BlastFutures/Blast-Futures-Exchange/ ↗
Version	Blast-Futures-Exchange: `fb2647caf781fbd167f3c71362f7d987335e7ee7`
Programs	• BfxVault.sol • BfxDeposit.sol • Bfx.sol • EIP712Verifier.sol • PoolDeposit.sol • Rabbit.sol
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:

Seunghyeon Kim
Engineer
seunghyeon@zellic.io

Jinseo Kim
Engineer
jinseo@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 7, 2024	Start of primary review period
May 8, 2024	End of primary review period

3. Detailed Findings

3.1. Missing yield-mode configuration on `setPaymentToken`

Target	Bfx		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The Bfx contract holds assets of customers. As WETH and USDB on Blast provide the yield to users holding the token, Bfx also has the features to claim this yield. Specifically, Bfx has the role `claimer`, which can trigger the claiming of the yield for the payment token. Accordingly, Bfx changes the yield mode of itself to `CLAIMABLE` in the constructor function, which allows assets and yield to be managed separately.

However, if the payment token of the contract is changed by the `setPaymentToken` function, the yield mode of the payment token is not changed.

Impact

The contract would receive the yield via rebasing of the asset if the payment token is changed. This issue breaks the features of the Bfx contract related to claiming.

Recommendations

Consider adding the logic that configures the yield mode of the new payment token.

Remediation

This issue has been acknowledged by Blast Futures.

Blast Futures stated that they do not expect to change the payment token unless it is required to do so due to the major event, and they would deploy the new contract if it breaks the functionality of the existing contract.

3.2. Minimum amount of `claimYield` does not work as intended

Target	Bfx		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The following is the code of the `claimYield` function:

```
uint256 constant HUNDRED_DOLLARS = 1e19;

// ...

function claimYield() external nonReentrant onlyClaimer {
    uint256 claimable = paymentToken.getClaimableAmount(address(this));
    if (claimable > HUNDRED_DOLLARS) {
        uint256 balanceBefore = paymentToken.balanceOf(address(this));
        paymentToken.claim(address(this), claimable);
        uint256 balanceAfter = paymentToken.balanceOf(address(this));
        require(balanceAfter > balanceBefore, "CLAIM_DIDNT_INCREASE_BALANCE");
        emit ClaimedYield(balanceAfter - balanceBefore);
    }
}
```

This code seems to enforce the minimum amount of claiming to be higher than a hundred dollars. However, this is misleading:

1. It is assumed that the payment token is USDB; however, it can be changed through the `setPaymentToken` function.
2. The constant `HUNDRED_DOLLARS` refers to 1e19, which would be 10 for the tokens with 18 decimals.

Impact

If the payment token of the contract is changed, this may affect the functionality of the `claimYield` function depending on the price of the new payment token.

This finding also documents the typo of the code affecting the logic.

Recommendations

Consider making the minimum amount of claiming to be configurable and changing the name or the value of the variable.

Remediation

This issue has been acknowledged by Blast Futures.

Blast Futures notified us that they were previously aware of the typo issue. Blast Futures also stated that they do not expect to change the payment token unless it is required to do so due to a major event, and they would deploy the new contract if it breaks the functionality of the existing contract.

3.3. Redundant code

Target	BfxVault, Rabbit, BfxDeposit, PoolDeposit		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The BfxVault and Rabbit contracts have the feature to manage the roles of users (admin, trader, treasurer). However, this feature is not used in these contracts.

Also, BfxDeposit and PoolDeposit contracts have this line of code unaffecting the behavior of the protocol:

```
function pooledDeposit(Contribution[] calldata contributions) external {
    // ...
    uint256 totalAmount = 0;
    // ...
    for (uint i = 0; i < contributions.length; i++) {
        // ...
        totalAmount += contribAmount;
        // ...
        require(totalAmount >= contribAmount, "INTEGRITY_OVERFLOW_ERROR");
        // ...
    }
    // ...
}
```

Starting with the Solidity version 0.8.x, the compiler performs an arithmethic overflow check in default; thus, the require statement is unnecessary.

Impact

The unused code does not have any impact on the behavior of the protocol.

Recommendations

Consider removing the unused code.

Remediation

This issue has been acknowledged by Blast Futures.

Blast Futures stated that they believe that the risk of this issue is negligible considering the cost of redeployment.

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. Owner of the contract cannot be changed

All contracts under the scope of this audit internally manage the owner of the contract. It has come to our attention that the owner address cannot be changed in all contracts under the scope of this audit.

Therefore, we recommend to manage the access of the owner address very carefully.

To our knowledge, a Gnosis Safe multi-signature wallet is the owner of PoolDeposit and Rabbit contracts on Ethereum Mainnet, and an EOA is the owner of Bfx, BfxDeposit, and BfxVault contracts on Blast.

4.2. Centralization risk

The contracts under the scope of this audit are the on-chain part of the project. The off-chain infrastructure is responsible to process the events emitted on the on-chain contracts and create a signature for withdrawals.

This implies the privilege that the off-chain infrastructure retains for the on-chain contracts. Notably, the off-chain infrastructure has a permission to withdraw all assets from Bfx and Rabbit contracts.

We believe that the off-chain infrastructure must be carefully developed and operated in order to implement the secure system. Also, we recommend to reduce the centralization risk where possible.

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: BfxDeposit.sol

Function: `individualDeposit(address contributor, uint256 amount)`

This function is for depositing the `paymentToken` to the `rabbit`.

Inputs

- `contributor`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the contributor to emit the `Deposit` event as.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must not be smaller than `MIN_DEPOSIT`, and the caller must have enough `paymentToken` to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to the `rabbit`.
  - ☑ Test coverage
- Increase `nextDepositId` when the transfer succeeds.
  - ☑ Test coverage

Negative behavior

- Revert when the amount is smaller than `MIN_DEPOSIT`.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

Function: `pooledDeposit(Contribution[] contributions)`

This function is for depositing the `paymentToken` to the `rabbit` with multiple contributors.

Inputs

- `contributions`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The length of the array must be smaller than `MAX_CONTRIBUTIONS`.
  - **Impact**: A set of contributions to treat as the deposited contributions.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to the `rabbit`.
  - ☑ Test coverage
- Increase the `nextDepositId` for each contribution.
  - ☑ Test coverage

Negative behavior

- Revert when the `contribution.amount` is smaller than `MIN_DEPOSIT`.
  - ☑ Negative test
- Revert when the `totalAmount` is not bigger than zero.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

5.2. Module: BfxVault.sol

Function: `stake(uint256 amount)`

This function emits the `Stake` event and transfers the `paymentToken` to the `bfx`.

Inputs

- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must not be smaller than `MIN_STAKE`, and the caller must have enough `paymentToken` to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to this contract.
  - ☑ Test coverage
- Increase `nextStakeId` when the transfer succeeds.
  - ☑ Test coverage

Negative behavior

- Revert when the amount is smaller than `MIN_STAKE`.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

5.3. Module: Bfx.sol

Function: `deposit(uint256 amount)`

This function is for depositing the `paymentToken`.

Inputs

- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must not be smaller than `MIN_DEPOSIT`, and the caller must have enough `paymentToken` to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to this contract.
  - ☑ Test coverage
- Increase `nextDepositId` when the transfer succeeds.
  - ☑ Test coverage

Negative behavior

- Revert when the amount is smaller than `MIN_DEPOSIT`.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

Function: `makeTransferFrom(address from, address to, uint256 amount)`

This function does ABI encoding to transfer the given amount of tokens from the given `from` to the given `to` and then calls `tokenCall`, which does a low-level call.

Inputs

- `from`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The address to transfer from.
- `to`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The address to transfer to.
- `amount`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The amount to transfer.

Function: `makeTransfer(address to, uint256 amount)`

This function does ABI encoding to transfer the given amount of tokens to the given address and then calls `tokenCall`, which does a low-level call.

Inputs

- `to`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The address to transfer.
- `amount`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The amount to transfer.

Function: `tokenCall(bytes data)`

This function does a low-level call on `paymentToken` with the given `data`.

Inputs

- `data`
  - **Control**: Completely controlled by the caller function.
  - **Constraints**: None.
  - **Impact**: The data to do the low-level call with.

Function: `withdrawTokensTo(uint256 amount, address to)`

This function transfers the `paymentToken` from this contract to the given address. This function is only for the contract owner.

Inputs

- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This contract must have enough tokens, and this value must be bigger than zero.
  - **Impact**: The amount to transfer.
- `to`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer.

Branches and code coverage

Intended branches

- Transfer the tokens to the given address.
  - ☑ Test coverage

Negative behavior

- Revert when the caller is not an owner.
  - ☑ Negative test

Function: `withdraw(uint256 id, address trader, uint256 amount, uint8 v, byte[32] r, byte[32] s)`

This function is for withdrawing the `paymentToken`, verifying the signature with given `v`, `r`, and `s`.

Inputs

- `id`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `processedWithdrawals` of given `id` must not be true and must be the signed `id` for the given signature.
  - **Impact**: An `id` to set as true on the `processedWithdrawals`.
- `trader`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be the signed `trader` for the given signature.
  - **Impact**: The address to send the `paymentToken`.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be the signed `amount` for the given signature, and the value must be bigger than zero.
  - **Impact**: The amount to transfer.
- `v`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.
  - **Impact**: A part of the signature to verify.
- `r`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.
  - **Impact**: A part of the signature to verify.
- `s`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.
  - **Impact**: A part of the signature to verify.

Branches and code coverage

Intended branches

- Verify the given signature.
  - ☑ Test coverage
- Send the `paymentToken` to the given `trader`.
  - ☑ Test coverage

Negative behavior

- Revert when it fails to verify the signature.

- ☑ Negative test
- Revert when the amount is not bigger than zero.
  - ☑ Negative test
- Revert when the given `processedWithdrawals` is already withdrawn.
  - ☑ Negative test

5.4. Module: PoolDeposit.sol

Function: `individualDeposit(address contributor, uint256 amount)`

This function is for depositing the `paymentToken` to the `rabbit`.

Inputs

- `contributor`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the contributor to emit the `Deposit` event as.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must be bigger than zero, and the caller must have enough `paymentToken` to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to the `rabbit`.
  - ☑ Test coverage
- Increase `nextDepositId` when the transfer succeeds.
  - ☑ Test coverage

Negative behavior

- Revert when the amount is not bigger than zero.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

Function: `pooledDeposit(Contribution[] contributions)`

This function is for depositing the `paymentToken` to the `rabbit` with multiple contributors.

Inputs

- `contributions`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: A set of contributions to treat as the deposited contributions.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to the `rabbit`.
  - ☑ Test coverage
- Increase the `nextDepositId` for each contribution.
  - ☑ Test coverage

Negative behavior

- Revert when the `contribution.amount` is not bigger than zero.
  - ☑ Negative test
- Revert when the `totalAmount` is not bigger than zero.
  - ☑ Negative test
- Revert when the transfer fails.
  - ☑ Negative test

5.5. Module: Rabbit.sol

Function: `deposit(uint256 amount)`

This function is for depositing the `paymentToken`.

Inputs

- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The caller must have enough `paymentToken` to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the `paymentToken` from `msg.sender` to this contract.
  - ☑ Test coverage

- Increase `nextDepositId` when the transfer succeeds.
  - ☑ Test coverage

Negative behavior

- Revert when the transfer fails.
  - ☑ Negative test

Function: `withdraw(uint256 id, address trader, uint256 amount, uint8 v, byte[32] r, byte[32] s)`

This function is for withdrawing the `paymentToken`, verifying the signature with given `v`, `r`, and `s`.

Inputs

- `id`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `processedWithdrawals` of given `id` must not be true and must be the signed `id` for the given signature.
  - **Impact**: An `id` to set as true on the `processedWithdrawals`.
- `trader`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be the signed `trader` for the given signature.
  - **Impact**: The address to send the `paymentToken`.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be the signed `amount` for the given signature, and the value must be bigger than zero.
  - **Impact**: The amount to transfer.
- `v`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.
  - **Impact**: A part of the signature to verify.
- `r`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.
  - **Impact**: A part of the signature to verify.
- `s`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid signature with given `id`, `trader`, and `amount`.

- **Impact**: A part of the signature to verify.

Branches and code coverage

Intended branches

- Verify the given signature.
  - ☑ Test coverage
- Send the `paymentToken` to the given `trader`.
  - ☑ Test coverage

Negative behavior

- Revert when it fails to verify the signature.
  - ☑ Negative test
- Revert when the amount is not bigger than zero.
  - ☑ Negative test
- Revert when the given `processedWithdrawals` is already withdrawn.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet and Blast.

During our assessment on the scoped RabbitX contracts, we discovered three findings. No critical issues were found. Two findings were of low impact and the remaining finding was informational in nature.

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.