Zellic

Resonate

Smart Contract Security Assessment

September 28, 2022

Prepared for:

Rob Montgomery

Revest Finance

Prepared by:

Ayaz Mammadov and Oliver Murray

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Revest Finance from July 18th to July 22nd, 2022.

Our general overview of the code is that it appears mechanically optimized and gas efficient, using queues to match issuers and purchasers.

We applaud Revest Finance for the documentation and the articles that detail in depth the inner workings of the Resonate project, explaining not only the mechanism but also the economic incentives of the market participants.

Zellic thoroughly reviewed the Resonate codebase to find protocol-breaking bugs and to find any technical issues outlined in the Methodology section (2.2) of this document.

Zellic met with the Revest Finance team to discuss their threat model. In our review we paid special attention to its pass through governance and reentrancy, oracle, and centralization risks.

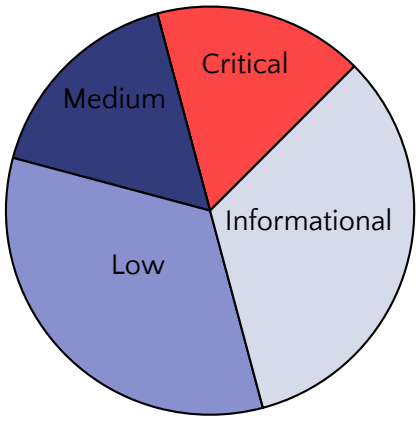
Due to the complexity of the protocol, we worked through many possible exchanges between issuers and purchasers across different pool configurations.

During our assessment on the scoped Resonate contracts, we discovered six findings. One of which was critical and has since been addressed by Revest. Of the remaining five findings, one was medium severity, two were of low severity, and the remaining were informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Revest Finance's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Resonate

From the creators of the Revest Protocol, Revest Finance is proud to introduce Resonate. Resonate intends to solve one of the biggest problems in DeFi by creating a marketplace for swapping up-front interest payments for future yields. Resonate uses pools to group traders by product attributes (locking terms, asset type, etc.) and clears orders using an automated queue-based matching system. The protocol is built on top of Revest and uses much of the existing infrastructure, including lockable FNFTs.

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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the asso-

ciated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible 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, upgradeability weaknesses, centralization risks, and so forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Resonate Contracts

Repository	https://github.com/Revest-Finance/Resonate
Versions	`6c3989dc8f5e2e7a0a60ad7792265ed246083219`
Programs	• AaveV2ERC4626 • AaveV2ERC4626Factory • ERC4626Factory • YearnWrapper_usdt • YearnWrapper • PoolSmartWallet • Resonate • ResonateHelper • ResonateSmartWallet • AddressLockProxy • OutputReceiverProxy
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Oliver Murray, Engineer
oliver@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 18, 2022	Start of primary review period
July 22, 2022	End of primary review period

3 Detailed Findings

3.1 Missing validation check in `createPool` can result in loss of user funds

- **Target**: Resonate
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: Critical

Description

The function `createPool( ... )` can be called on an already existing pool when `additionalRate > 0 && lockupPeriod == 0`. The check for a preexisting pool in `initPool` only addresses the case of `(lockupPeriod ≥ MIN_LOCKUP && additionalRate == 0)` by using the following check `require(pools[poolId].lockupPeriod == 0, 'ER002')`.

Impact

A malicious user could recreate an already existing pool. This would reset the `PoolQueue( ... )`, which tracks the positions in the queue of the consumer and producer orders. These orders would effectively be taken out of the matching algorithm. If the pool had only processed a limited number of orders, the previous orders could easily be overwritten and no longer modified using `modifyExistingOrder( ... )`. Once overwritten, there would be no way to retrieve the funds from the PoolSmartWallet.

Recommendations

Expand the `require` checks in `initPool( ... )` to the following:

```
function initPool(
        address asset,
        address vault,
        uint80 rate,
        uint80 _additional_rate,
        uint32 lockupPeriod,
        uint packetSize
    ) private returns (bytes32 poolId) {
        poolId = getPoolId(asset, vault, rate, _additional_rate,
    lockupPeriod, packetSize);
```

```
        require(pools[poolId].lockupPeriod == 0 && pools[poolId].
    addInterestRate == 0, 'ER002');
```

Remediation

This finding was remediated by Revest in commit `f19896868dd2be5c745c66d9d75219f6b04a593c`.

3.2 Failure to cancel orders in `modifyExistingOrder`

- **Target**: Resonate
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

Producers are not able to cancel and recover funds on queued orders using `modifyExistingOrder( ... )` for cross-asset pools. Calling `submitProducer( ... )` always sets `shouldFarm = false` and `order.depositedShares > 0` using the oracle price:

```
if(shouldFarm) {
    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    order.depositedShares = IERC4626(vaultAdapter).deposit(amount,
    getAddressForPool(poolId)) / order.packetsRemaining;
} else {
    IERC20(asset).safeTransferFrom(msg.sender, getAddressForPool(poolId),
     amount);
}
```

However, `modifyExistingOrder( ... )` has missing checks and assumes the orders were deposited in the vault asset instead of the pool asset:

```
if (order.depositedShares > 0) {
    getWalletForPool(poolId).withdrawFromVault(amountTokens, msg.sender,
    vaultAdapters[pool.vault]);
} else {
    getWalletForPool(poolId).withdraw(amountTokens, pool.asset, msg.
    sender);
}
```

The attempt to withdraw from the vault asset from the pool wallet will fail. Fortunately, there are no vault assets in the pool wallet to exploit because all vault assets are sent to the FNFT wallet (ResonateSmartWallet) when orders are matched. However, a producer would not be able to retreive the funds of their order.

It should be noted that attempting to fix this bug by only directing `modifyExistingOrder( ... )` to retreive the pool asset instead of the vault asset will result in a critical exploit. This is because `submitProducer( ... )` accounts for the price of the vault asset while `modifyExisitngOrder( ... )` does not.

For example, the producer deposits `amount` of pool assets and gets credited packets equal to `amount/ producerPacket`:

```
...
sharesPerPacket = IOracleDispatch(oracleDispatch[vaultAsset][pool.asset])
    .getValueOfAsset(vaultAsset, pool.asset, true);
producerPacket = getAmountPaymentAsset(pool.rate * pool.packetSize/
    PRECISION, sharesPerPacket, vaultAsset, vaultAsset);
...
producerOrder = Order(uint112(amount/ producerPacket), sharesPerPacket,
    msg.sender.fillLast12Bytes());
```

Through `getAmountPaymentAsset( ... )` the `producerPacket` scales linearly with the vault price. However, if the producer tries to later modify their order, there is no adjustment from the number of packets to the amount of pool asset:

```
...
if (isProvider) {
    providerQueue[poolId][position].packetsRemaining -= amount;
  } else {
    consumerQueue[poolId][position].packetsRemaining -= amount;
}
...
uint amountTokens = isProvider ? amount * pool.packetSize * pool.rate /
    PRECISION : amount * pool.packetSize;
```

If `vault price > 1` the producer will not be refunded a sufficient amount of assets for the reduction in packets. This is because `submitProducer( ... )` scales down the packets by the vault price, while `modifyExistingOrder( ... )` does not commensurately scale up the amount of pool asset per packet.

If `vault price < 1` the producer will be refunded an excessive amount of assets for the reduction in packets. This is because `submitProducer( ... )` scales up the packets by the vault price, while `modifyExistingOrder( ... )` does not commensurately scale down the amount of pool asset per packet.

Impact

Order cancelling for producers would be nonoperational.

Recommendations

The following changes should be made to `modifyExistingOrder( ... )`: (1) withdrawl the pool asset for cross-asset producer orders and (2) use the price of the vault asset at the time the order was submitted to correctly calculate `amountTokens`.

Remediation

This finding was remediated by Revest in commit `fc3d96d91d7d8c5ef4a65a202cad18a3e86a3d09`.

3.3 Failed approval check in `calculateAndClaimInterest`

- **Target**: ResonateSmartWallet
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The allowance check for token transfer approval always fails in `calculateAndClaimInterest( ... )`:

```
) public override onlyMaster returns (uint interest, uint sharesRedeemed)
     {
    IERC4626 vault = IERC4626(vaultAdapter);
    if(IERC20(vaultToken).allowance(address(this), vaultAdapter) <
    interest) {
        IERC20(vaultToken).approve(vaultAdapter, type(uint).max);
    }
```

The if statement will always fail because `interest` has not been initialized from zero.

Impact

Minimal - other functions in ResonateSmartWallet will be called that also set the token transfer approval to `max`. In the worst case scenario, the very first producer order will be delayed in claiming interest until the first consumer order reclaims their principal.

Recommendations

Change `interest` to `totalShares` in the `if` control statement.

Remediation

This finding was remediated by Revest in commit `6b1b81f6c0310297f5b6cd9a258b99e43c61b092`.

3.4 Incorrect asset tracking in modifyExistingOrder

- **Target**: Resonate
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

For cross-asset pools, calling submit `submitProducer( ... )` always sets `shouldFarm = false` and `order.depositedShares > 0` using the oracle price. Orders are then enqueued with the pool asset:

```
if(shouldFarm) {
    IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
    order.depositedShares = IERC4626(vaultAdapter).deposit(amount,
    getAddressForPool(poolId)) / order.packetsRemaining;
} else {
    IERC20(asset).safeTransferFrom(msg.sender, getAddressForPool(poolId),
     amount);
}
```

However, `modifyExistingOrder( ... )` has missing checks and assumes the orders were deposited in the vault asset:

```
if (order.depositedShares > 0) {
    getWalletForPool(poolId).withdrawFromVault(amountTokens, msg.sender,
    vaultAdapters[pool.vault]);
} else {
    getWalletForPool(poolId).withdraw(amountTokens, pool.asset, msg.
    sender);
}
```

Impact

An attacker could spam `submitProducer( ... )` and `modifyExistingOrder( ... )` to convert pool assets to vault assets at a rate of `1:1`. This could be financially lucrative as there are no ways to shut down the protocol or pull other users funds. It would also disrupt of the balance of the cross-asset pair and hence the potential operation of the pool.

Recommendations

Include logic to enure the vault and pool assets are correctly tracked in cross asset pools.

Remediation

Revest has implemented the following solution in committ 00000000000:

```
if (order.depositedShares > 0 && IERC4626(vaultAdapters[pool.vault]).
    asset() == pool.asset)
```

We find their remediation adequately addresses the concerns of this finding.

3.5 Missing validation check in `proxyCall` filter can allow dangerous calls

- **Target**: ResonateSmartWallet
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `proxyCall` function has checks to ensure no calls made to it result in a decrease of capital. However, it has incomplete checks to ensure there are no calls made that could result in a future decrease of capital. For example, it currently includes a filter for `approve` but none for newer functions like `increaseAllowance`.

Impact

The `proxyCall` function can only be called by the sandwich bot. In the case of a compromise or a security incident involving keys, the lack of the requisite checks could result in a loss of funds.

Recommendations

We recommend adding a check for the `increaseAllowance` function selector. The use of an adjustable white list or black list to control allowed functions would provide additional flexibility for unforseen risky functions. The management of the white list/black list should be delegated to another administrative account to limit centralization risk.

Remediation

Revest has indicated this will be resolved at deployment-time by modifying the deployment-script to include the `increaseAllowance` function signature.

3.6 Centralization risk

- **Target**: Project Wide
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Low

Description

At the end of deployment and configuration of the AddressLockProxy, OutputReceiverProxy, ResonateHelper, and Resonate, ownership is primarily concentrated in a single account. However, a specially designated sandwich bot is able to access the `proxyCall( ... )` and `sandwichSnapshot` functions in the ResonateHelper. These functions cannot move funds outside of the system but can move the location of funds within the system for the purpose of snapshot voting. When new pools are added to resonate they are created along with their own ResonateSmartWallet and PoolSmartWallet contracts. These wallets can only be accessed by Resonate. There are no owners of the `ERC4626` adapters used to interface between Resonate and the vaults.

In general, the owner of Resonate cannot stop the protocol or withdraw funds other than through regular use of the protocol. However, they are in control of the address of the oracle. By manipulating the price of the oracle they could grossly inflate the number of packets a producer order is entitled to and profit from matches with consumer orders (more in the discussion on oracle risk).

The protocol relies heavily on the proper functioning of several external vaults. Under the current scope of this audit these include Aave and Yearn. Compromise of these vaults could break the system and result in loss of funds. This is viewed as an acceptable and necessary risk.

Resonate also relies on several key contracts in the Revest ecosystem. These include a registry that returns the address of Revest and the FNFT Handler. Compromise of this registry could direct Resonate to interact with compromised contracts. Furthermore, compromise of Revest or the FNFT handler could break the protocol or result in loss of funds. For example, Revest is responsible for calling critical functions in Resonate for claiming interest and principal. The burning of FNFTs is handled by Revest, and the FNFT handler and its compromise could potentially result in repeated claiming of interest and/or principal.

Impact

Control of Resonate is heavily concentrated in a single account; however, compromise of this account presents limited vectors for exploitation. A compromised owner account could alter the price oracle to one in their control and use this to exploit the

system for financial gain.

The compromise of the sandwich bot could result in abuse of `proxyCall` and `sandwichSnapshot`, which could disrupt the proper functioning of the protocol.

Recommendations

The use of a multisignature address wallet can prevent an attacker from causing economic damage in the event a private key is compromised. Timelocks can also be used to catch malicious executions. It should be verified that this practice is being followed for not just the core Resonate contracts (including the sandwich bot) but also the other contracts it interacts with listed above.

The oracle should be carefully set to a trusted source such as ChainLink or an alternative that uses a sufficiently long TWAP. Care needs to be taken in ensuring the price oracle cannot be manipulated through flash loans or other means of attack.

Remediation

Revest has provided a highly detailed response which adequately addresses our concerns around the access management of critical contracts. Their procedures for managing centralization risk include the following:

- Resonate will use, at a minimum, a 3 of 5 multisig. No more than a simple majority will be core team members, the remainder will be drawn from the community. The members of the Resonate multisig will have no more than two members overlapping with the Revest multisig.
- Sandwich bot access will initially align with Resonate access.
- Revest currently uses a 3 of 7 mutlisig. This will be upgraded to a 4 of 7 soon.
- The registry is currently controlled by a multisig.
- A multisig will be used to control the oracle systems.
- The FNFT handler is immutable.
- An individual will posesses no more than one key on a given multisig. In general the use of hardware wallets is either mandated (Resonate) or encouoraged (Revest, non-officers).
- As progressive decentralization occurs, control over many of the contracts in the Revest-Resonate ecosystem will be migrated to intermediary contracts/DAOs.

3.7 Missing correct event information

- **Target**: Resonate
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The event triggered in `dequeue( … )` does not contain the dequeued order information but rather the order information of the remaining head.

```
if(isProvider) {
    delete providerQueue[poolId][qm.providerHead++];
    emit DequeueProvider(poolId, msg.sender, providerQueue[poolId][qm.
    providerHead]);
} else {
    delete consumerQueue[poolId][qm.consumerHead++];
    emit DequeueConsumer(poolId, msg.sender, consumerQueue[poolId][qm.
    consumerHead]);
}
```

Impact

Providing the removed order in the event would be more useful for event listeners.

Recommendations

Include the correct order information in the event dequeue event.

Remediation

This finding was remediated by Revest in commit `9177c788cb2f3304b16f1583696794f24e1a0a92`.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Oracle attacks

If an attacker got control of the price oracle, they could pass a low price to `sharesPerPacket` during a call to `submitProducer(…)`:

```
sharesPerPacket = IOracleDispatch(oracleDispatch[vaultAsset][pool.asset])
    .getValueOfAsset(vaultAsset, pool.asset, true);
```

The depressed price would drive up the number of packets of vault shares for interest claiming.

```
…

producerPacket = getAmountPaymentAsset(pool.rate * pool.packetSize/
    PRECISION, sharesPerPacket, vaultAsset, vaultAsset);…

producerOrder = Order(uint112(amount/ producerPacket), sharesPerPacket,
    msg.sender.fillLast12Bytes());
```

They would get matched with a higher amount of underlying vault principal for the same dollar amount of pool asset deposited, allowing them to earn excessive interest. Similar to the points in the section on centralization risk, this attack vector is best managed by 1) using a multisig to set the price oracle address and 2) using a reliable price oracle such as ChainLink.

Revest acknowledges this risk. They emphasize that oracle systems require administrative controls and indicate the use of multisigs and timelocks as eventual control measures.

4.2 Reentrancy

There are sections of code that do not follow the checks-interactions-effects design pattern used to prevent reentrancy attacks. There are limited possibilities for exe-

cution control to pass outside of safety. Furthermore, it appears that in most cases other system variables (outside of those accounting for balances before and after fund transfer) are used to prevent reentering the contract. However, there is no reason not to use the `nonReentrant` modifier to prevent any possibility of reentry.

Revest has applied the `nonReentrant` modifier on all pertinent functions in Resonate, ResonateSmartWallet, and PoolSmartWallet - commit `b81a509b41524c896f8bfa75785b554496e16080`.

4.3 Code maturity

Follow the adopted conventions for internal and private variables

Best practices for solidity development use the _ to prefix both internal and private variables and functions.

This was addressed by Revest in commit `b81a509b41524c896f8bfa75785b554496e16080`.

Reliance on integer underflow reversion

In numerous places reversion by integer underflow is used as an implicit check that withdraw amounts do not exceed available funds. For example, this happens in `receiveResonateOutput( ... )` as there is no check made on the function parameter `quantity`. It also happens in `modifyExistingOrder` as there is no check on `amount`. Including explicit checks on these parameters would send clear messages to protocol users under transaction failure and improve the overall user experience.

Revest indicated they may address this in the future.

Confusing variable names

In general the variables are intuitively named. However, the method `getAddressForFNFT(bytes32 fnftId)` in ResonateHelper takes an `fnftId` parameter but is in fact passed a `poolId`. This can create some developer confusion because the variable name `fnftId` is also used in Resonate to denote the ID of the FNFT. We suggest Revest change the parameter name from `fnftId` to `poolId`.

Revest indicated this has been addressed.

Clear comments

At the time of audit the Resonate project is still a work in progress. There are many comments left for other developers that take the form of questions and to-do lists. There is also a general lack of good-quality comments that would be useful for some-

one not intimately familiar with the code base. This applies to both the core contracts and their interfaces. For example, the following comment indicates a work in progress but also points to an unused variable:

```
function maxDeposit(address _account)
    public
    view
    override
    returns (uint256)
{
    _account; // TODO can acc custom logic per depositor
    VaultAPI _bestVault = yVault;
    uint256 _totalAssets = _bestVault.totalAssets();
    uint256 _depositLimit = _bestVault.depositLimit();
    if (_totalAssets >= _depositLimit) return 0;
    return _depositLimit - _totalAssets;
}
```

Revest has made considerable improvements to the inline documentation.

Unused resources

There are potentially unused resources in the project; for example, `FullMath` is never used for `uint256` in Resonate.

Revest has removed the unused library - `6b1b81f6c0310297f5b6cd9a258b99e43c61b092`.

Control variables and abstraction

Using values of process variables like `depositedShares` to indicate pool and order configurations is challenging to read. And furthermore, as we have seen in these report findings, it is error prone. Revest should consider adding another layer of abstraction to more clearly illustrate pool and order configurations.

4.4 Composability

As indicated in the finding on centralization risk, the Resonate protocol relies heavily on composability. It is therefore important to note that the improper functioning of any of the composable contracts lying outside of the scope of this audit is likely to cause considerable failure in Resonate. These contracts include the investment vaults (Aave and Yearn), the oracle price sources, Revest, the Revest registry, and the Revest

FNFT handler.

It is important to note that similar suggestions and observations presented in the centralization risk finding also apply to these dependencies as well.

5 Audit Results

At the time of our audit, the code was not deployed to mainnet evm.

During our audit, we discovered 6 findings. One of which was critical, one of which was medium risk, and the remaining four were split evenly between low risk and informational.

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.