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 a second audit for Revest Finance from July 25th to July 27th, 2022.

This second audit covered the same code base as the audit performed from July 18th to July 22nd, 2022. This second audit included new contracts and some changes to contracts reviewed in the previous audit.

All findings presented here reflect only the new code. Please refer to the previous report for findings that still exist in this code base but that were present at the time of last review. All discussion points that still broadly apply remain.

Our general overview of the code is unchanged. 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) TODOcould this number be hyperlinked? 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.

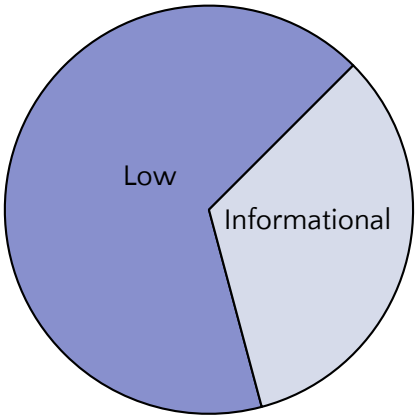
Due to the complexity of the protocol, we worked through many possible exchanges between issuers and purchasers across different pool configurations.

During our assessment on the scoped Resonate contracts, we discovered three findings. Two of which were of low severity, the remaining was 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. Notes that still apply to the code based from the previous audit have been left as is for local completeness.

Breakdown of Finding Impacts

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



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

Previous report versions `6c3989dc8f5e2e7a0a60ad7792265ed246083219`

Current versions `4633085640f777ca0eeb894189d331f1c4c70a92`

Programs

- AaveV2ERC4626
- AaveV2ERC4626Factory
- ERC4626Factory
- YearnWrapper_usdt
- YearnWrapper
- PoolSmartWallet
- Resonate
- ResonateHelper
- ResonateSmartWallet
- AddressLockProxy
- OutputReceiverProxy
- MasterChefAdapter
- MasterChefV2Adapter

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 25, 2022	Start of primary review period
July 27, 2022	End of primary review period

3 Detailed Findings

3.1 Inconsistent interest calculations in Resonate

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

Description

In order to convert from the amount of underlying assets to shares, `claimInterest` uses `previewWithdraw( ... )`:

```
function previewWithdraw(uint256 assets)
    public
    view
    override
    returns (uint256)
{
    return convertToShares(assets);
}
[ ... ]
function convertToShares(uint256 assets)
    public
    view
    override
    returns (uint256)
{
    return (assets * yVault.totalSupply()) / getFreeFunds();
}
```

However, `batchClaimInterest` uses the following to convert from assets to shares:

```
uint shareNormalization = vault.totalSupply() * PRECISION / vault.
    totalAssets();
[ ... ]
uint totalSharesUnderlying = shareNormalization * amountUnderlying /
    PRECISION;
```

When interacting with the yearn vault, this results in a difference in interest calculations between the two functions - `claimInterest( ... )` uses `getFreeFunds()` in the divisor while `batchClaimInterest` uses `vault.totalAssets();`.

Impact

The interest calculation of `batchClaimInterest` would be inflated over `claimInterest` by the difference in denominator in the asset-to-share conversion.

Recommendations

The difference is probably not intended by developers. It would be better to leverage composability and to use `previewWithdraw( ... )` in `batchClaimInterest`. This would ensure `batchClaimInterest` is consistent with `claimInterest` across the different adapter implementations.

Remediation

Revest has followed the recommendation and is using `previewWithdraw( ... )` in commit `da89259c00235fdab7edbcb638ec04f3e360ef48`.

3.2 Incomplete whitelist and blacklist functionality in Resonate-Helper

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

Description

Once a `fxSelector` has been added to the whitelist, it cannot later be blacklisted.

For example, if the function has not been blacklisted it can be set in the whitelist:

```
function whiteListFunction(uint32 selector) external onlySandwichBot
    glassUnbroken {
    require(!blackListedFunctionSignatures[selector], “ER030”);
    whiteListedFunctionSignatures[selector] = true;
}
```

And if the function has been whitelisted, it can still be blacklisted:

```
function blackListFunction(uint32 selector) external onlySandwichBot
    glassUnbroken {
    blackListedFunctionSignatures[selector] = true;
}
```

However, if a function has been whitelisted and is then blacklisted, it will still pass the validation check in `proxyCall(…)` because function logic only requires the `fxSelector` to exist in the whitelist:

```
function proxyCall(bytes32 poolId, address vault, address[] memory
    targets, uint[] memory values, bytes[] memory
    calldatas) external onlySandwichBot glassUnbroken {
    for (uint256 i = 0; i < targets.length; i++) {
        require(calldatas[i].length ≥ 4, “ER028”); //Prevent calling
    fallback function for re-entry attack
        bytes memory selector = BytesLib.slice(calldatas[i], 0, 4);
        uint32 fxSelector = BytesLib.toUint32(selector, 0);
        require(whiteListedFunctionSignatures[fxSelector], “ER025”);
    }
```

```
    ISmartWallet(_getWalletForFNFT(poolId)).proxyCall(vault, targets,
    values, calldatas);
}
```

Impact

If the sandwichbot were to mistakenly set a dangerous function (or a function that later turned out to be dangerous) to the whitelist they would not be able to later block that function from being passed to `proxyCall( ... )`.

Recommendations

Include logic to blacklist previously whitelisted functions. The blacklist should be immediately set to include `increaseAllowance` and `approve` as these functions can be used to increase spending allowance, which can trigger transactions that would pass the balance checks on `proxyCall( ... )` in ResonateSmartWallet.

Remediation

Revest has added in the functionality that would allow for blacklisting of previously whitelisted functions in commit `f95f9d5ac4ac31057cef185d57a1a7b03df5f199`. The functions `increaseAllowance` and `approve` have been added to the blacklist in commit `f2428392e0ce022cd6fde9cf41e654879c03119c`.

3.3 Missing claim event in `batchClaimInterest`

- **Target**: Resonate
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

There is no event emitted indicating interest has been claimed during a call to `batchClaimInterest`.

Impact

Providing information on interest claimed in batches would be useful to downstream listeners.

Recommendations

Include an event announcing interest claimed during batch calls, ideally at the pool level.

Remediation

Revest has followed the recommendation and added the event in commit `28261625047b025bf1baa17fcbb4c8b114878590`.

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.

4.2 Reentrancy

The functions `batchClaimInterest( … )` in Resonate and `redeem( … )` in ResonateSmartWallet have been added since the last review and do not include the `nonReentrant` modifiers. While these functions do not appear to be reentrancy concerns, we advise Revest similarly apply the `nonReentrant` modifiers to these functions.

The `nonReentrant` modifier has been added to `redeem( … )` in commit `47d6c1393264ea`

71cad1fa8a7ae8e24869c5ae37 and to batchClaimInterest(...) in commit da89259c00235fdab7edbcb638ec04f3e360ef48.

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 someone 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 'L!\TODO{I think this finding is from the previous report, so I might suggest referring to that report here}, 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 cen-

tralization 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 three findings. Two of which were low, the remaining was 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.