Zellic

Revest Finance

Smart Contract Security Assessment

April 25, 2022

Prepared for:

Rob Montgomery

Revest Finance

Prepared by:

Jasraj Bedi, Ayaz Mammadov and Jacob Farrell

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 Introduction

1.1 About Revest Finance

Revest Finance proposes a new protocol for the packaging, transfer, and storage of fungible ERC-20 tokens as non-fungible tokenized financial instruments by leveraging the ERC-1155 Non-Fungible Token (NFT) standard for ease of access and universality of commerce.

1.2 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, etc. 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, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We

look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

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 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, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Revest Finance Contracts

Repository	https://github.com/Revest-Finance/RevestContracts
Versions	`20b5041445440f6e985448e324e534f9eeda8f32`
Programs	• RevestContracts
Type	Solidity
Platform	EVM-compatible

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 3 person-week. The assessment was conducted over the course of 2 calendar weeks.

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

Jacob Farrell, Engineer
jacob@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

April 11, 2022	Kick-off call
April 11, 2022	Start of primary review period
April 22, 2022	End of primary review period
May 2, 2022	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

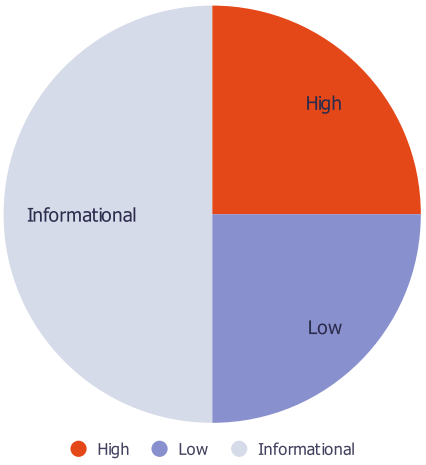
2 Executive Summary

Zellic conducted an audit for Revest Finance from Aprill 11th, 2022 to April 22nd, 2022 on the scoped contracts and discovered 4 findings. Fortunately, no critial issues were found. Of the 4 findings one was of high impact, one was of low impact and the remaining findings were informational in nature.

The overall summary of the Revest Finance project was a well-maintained and kept code base which we compliment. Due to the nature of the project and the large amount of user-input, opportunites for re-entrancy and user-controlled execution were frequent. This is especially notable because of how the last exploit which targeted Revest Finance worked. This forced us to be extremely attentful of user-controlled executions and other re-entrancy points, but is also mitigated by the fact that all external functions are marked `nonReentrant`.

Breakdown of Finding Impacts

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



3 Detailed Findings

3.1 Potential loss of funds caused by fake rebases

- **Target**: TokenVault.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The calculation of `lastMul` to account for rebase tokens is incorrect and can lead to devaluation of user funds deposited in the vault

```
function updateBalance(uint fnftId, uint incomingDeposit) internal {
    ...
    if(asset != address(0)){
        currentAmount = IERC20(asset).balanceOf(address(this));
    } else {
        // Keep us from zeroing out zero assets
        currentAmount = lastBal;
    }
    tracker.lastMul = lastBal == 0 ? multiplierPrecision :
    multiplierPrecision * currentAmount / lastBal;
    ...
    }
```

The `TokenVault` supports rebase tokens with a dynamic supply to achieve a certain economic goal, such as pegging a token to a certain asset.

In `TokenVault`, we can see that the `currentAmount` is the balance of the `TokenVault` divided by `lastBal`. This checks whether the asset has rebased since the last interaction, signaling an increase or decrease in supply.

However, an attacker may transfer ERC20 tokens directly to the vault, inflating `currentAmount`, leading to an inflated `lastMul`, thus emulating a rebase. The deposit with inflated `lastMul` would be devalued when `lastMul` is reset back in the next `updateBalance` call.

PoC

A sample proof of concept can be found here.

The output is as follows:

```
Minted one FNFT with id -> 0
Current value of FNFT-0 is 10
Transferred 10 tokens to fake a rebase
Minted another FNFT with id -> 1 and 100 depositAmount
The value should be 100
But the value is 50
```

The relevant output shows 2 different mints and the effect of a direct transfer on the last mint, reducing the value of the minted FNFT to 50 instead of the original 100.

Impact

The victim minting a FNFT following the fake rebase action permenantly loses funds. This poses a very large griefing vector for Revest.

Recommendations

Alter the logic as to properly account for Rebase Tokens.

Remediation

The Revest team has fixed this issue by proposing a move to a new and improved TokenVaultV2 design, and by deprecating the handling of rebase tokens in TokenVault.

3.2 createFNFT should not be public

- **Target**: TokenVault.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: N/A
- **Impact**: N/A

description

```
function createFNFT(uint fnftId, IRevest.FNFTConfig memory fnftConfig,
    uint quantity, address from) external override {
        ...
}
```

`createFNFT` should not be external, as all of its' internal function calls are restricted to `onlyRevestController`.

Impact

The issue currently has no security impact, but developers should avoid loose access controls when possible as it may lead to unexpected behavior or bugs in the future.

Recommendations

Add the `onlyRevestController` modifier to `createFNFT` to restrict access control.

Remediation

The issue has been acknowledged by Revest team.

3.3 Batched mints can be rejected by a single recipient

- **Target**: FNFTHandler.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

```
function mintBatchRec(address[] calldata recipients, uint[] calldata
    quantities, uint id, uint newSupply, bytes memory data) external
    override onlyRevestController {
    supply[id] += newSupply;
    fnftsCreated += 1;
    for(uint i = 0; i < quantities.length; i++) {
        _mint(recipients[i], id, quantities[i], data);
    }
}
```

A batched mint from `mintBatchRec` is suspectible to being cancelled by a single recipient failing the ERC-1155 `AcceptanceCheck`

Impact

Gas is lost and other willing recipients do not receive the FNFTs and the batched mint execution has to be repeated.

Recomendations

- Execute the batched mint in a try catch loop and refund if a mint fails.
- If intended, document this behaviour.

Remediation

The issue has been acknowledged by the Revest team and a fix is pending.

3.4 Codebase maturity

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

There are several areas in the project that merit an improvement in documentation, areas that have no documentation at all and especially areas where re-entrancy should be considered, such that any future contributors are more aware of potential security risks.

For example, `createLock` accepts many user-controlled inputs such as `Oracle` and `Asset` but this is not documented anywhere as a possible re-entrancy point. `withdrawToken` mentions a callback to the output receiver but only on the line right before and with no extra emphasis.

Unfortunately, we were unable to run the test suite in its complete capacity, although it seems there has been an extensive effort to test functions even including a poc for the previously targeted exploit.

Impact

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs, should the code need to be modified later on. In general, lack of documentation impedes auditors and external developers from reading, understanding, and extending the code. The problem will also be carried over if the code is ever forked or re-used.

Recommendations

Clearly document functions where there may be a re-entrancy risk or document functions where users have control over provided arguments through which calls will be executed.

Remediation

The issue has been acknowledged by the Revest team and a fix is pending.

4 Discussion

In this section, we discuss miscellaneous interesting observations during the audit that are noteworthy and merit some consideration.

Revest's approach to standardizing the storage of ERC20 tokens using FNFTs is very commendable and addresses many current problems in the DeFi space.

We did a preliminary review of TokenVaultV2, which fixes a lot of issues with the current TokenVault. Instead of storing assets in a single contract, the new vault leverages `CREATE2` to create per-NFT vaults. This simplifies a lot of internal accounting, and is overall a great step in creating a better security posture by reducing the attack surface. As it isn't production ready, few feature such as migrations are implemented however they are not initialized properly, rendering them dysfunctional.

In the fnft-migrations branch, we noted that the `_beforeTokenTransfer` calls back to `IOutputReceiverV4`. But, it will not callback on `mint` operations due to the `from` $\neq$ `0` check although it will callback on `burn` operations. However the callback on `burn` operations is only on applied to the first `burn` if a batch is `burned`. While none of this poses an explicit security risk, this makes composability much harder and we therefore recommend documenting these effects or changing them for simplicity's sake.

In MetadataHandler.sol, several setter functions (`setTokenURI` and `setRenderTokenURI`) reponsible for setting the rendering and Token URI are lacking access controls. It is worth noting that they are not crucial to overall logic of the system, except for the frontend.

Re-entrancy points

User control of execution mostly comes from the fact that users can have full control over `asset` and `pipeToContract` in `FNFTConfig`. Users control `ValueLock`'s `oracle` and `AddressLock`'s `trigger` which can also lead to external, user-controlled calls.

Finally, it is important to keep in mind a callback is called on `to` whenever an `ERC1155` is minted. Although there are so many re-entrancy points, they're not directly exploitable as every external function is marked `nonReentrant`.

Revest.sol

Revest.sol:{333, 354} - user controls `fnftConfig.asset`

```
333: IERC20(fnftConfig.asset).safeTransferFrom(_msgSender(),
    addressesProvider.getAdmin(), totalERC20Fee);
...
354: IERC20(fnftConfig.asset).safeTransferFrom(_msgSender(), vault,
    totalQuantity * fnftConfig.depositAmount);
```

Revest.sol:136 - user controls `trigger`

```
136: IAddressLock(trigger).createLock(fnftId, lockId, arguments);
```

Revest.sol:{203, 248} - user controls `pipeToContract`

```
203: IOutputReceiverV3(config.pipeToContract).handleTimelockExtensions(
    fnftId, endTime, msg.sender);
...
248: IOutputReceiverV3(fnft.pipeToContract).handleAdditionalDeposit(
    fnftId, amount, quantity, msg.sender);
```

Revest.sol:358-362 - ERC1155 `_mint` executes callback on `to`

```
358: if(!isSingular) {
359:     getFNFTHandler().mintBatchRec(recipients, quantities, fnftId,
    totalQuantity, '');
360: } else {
361:     getFNFTHandler().mint(recipients[0], fnftId, quantities[0], '');
362: }
```

LockManager.sol

LockManager.sol:{60, 158} - user controls `lock.valueLock.oracle`

```
60: IOracleDispatch oracle = IOracleDispatch(lock.valueLock.oracle);
...
158: IOracleDispatch oracle = IOracleDispatch(lock.valueLock.oracle);
```

LockManager.sol:{121, 143} - user controls `lock.addressLock`

```
121: IAddressLock(addLock).isUnlockable(fnftId, lockId))
...
143: IAddressLock(lock.addressLock).isUnlockable(fnftId, fnftIdToLockId[
    fnftId])
```

TokenVault.sol

TokenVault.sol:{60, 104, 110, 229} - user controls `asset`

```
60: currentAmount = IERC20(asset).balanceOf(address(this));
...
104: IERC20(asset).safeTransfer(user, withdrawAmount);
...
110: IERC20(asset).safeTransfer(fnft.pipeToContract, withdrawAmount);
...
229: currentAmount = IERC20(fnfts[fnftId].asset).balanceOf(address(this))
    ;
```

TokenVault.sol:{114, 224} - user controls `pipeToContract`

```
114: IOutputReceiver(pipeTo).receiveRevestOutput(fnftId, asset, payable(
    user), quantity);
...
224: return IOutputReceiver(fnfts[fnftId].pipeToContract).getValue((
    fnftId));
```

FNFTHandler.sol (in the FNFT-migration branch)

FNFTHandler.sol:{106, 117} - user controls `pipeToContract`

```
106: IOutputReceiverV4(config.pipeToContract).onTransferFNFT(ids[0],
    operator, from, to, amounts[0], data);
...
117: IOutputReceiverV4(config.pipeToContract).onTransferFNFT(ids[i],
    operator, from, to, amounts[i], data);
```