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.

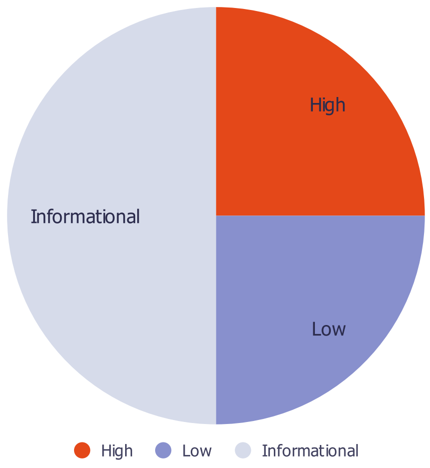
2 Executive Summary

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 Griefing opportunity may cause users to lose funds

- **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 certain economic goals, such as pegging a token to an 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.

Proof of Concept

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 PoC mints two FNFTs. The first one proceeds as normal. Then, tokens are transferred directly to the vault. This transfer emulates a "fake" rebase. As a result, when the second FNFT is minted, it has value 50 rather than the correct value of 100.

Impact

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

Recommendations

Alter the logic 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 Certain functions' access controls are unnecessarily lax

- **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 {
        ...
}
```

The function `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 abide by the principle of least privilege. Limiting a contract's attack surface is a crucial way to mitigate future risks and reduces the overall likelihood and severity of compromises.

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 susceptible to being cancelled by a single recipient failing the ERC-1155 `AcceptanceCheck`

Impact

Gas is wasted, and other willing recipients do not receive the FNFTs. The batched mint execution has to be retried.

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 Unclear documentation of security risks

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

There are several areas in the project which should be documented more thoroughly, and some areas have no documentation at all. This is especially true in areas where re-entrancy must be considered, so that future contributors are alerted to potential security hazards.

For example, the function `createLock` accepts several user-controlled inputs like `Oracle` and `Asset`, but this is not documented anywhere as a possible re-entrancy point. The function `withdrawToken` mentions a callback to the output receiver. However, the warning is placed deep in the function body, and with no special emphasis.

Impact

Code maturity is critical 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. It is imperative to clearly and prominently highlight any potential security hazards, such as re-entrancy, as these hazards could be fatal.

Recommendations

Clearly document functions where there is the risk of re-entrancy, as well as functions where user-controlled inputs are processed.

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 novel.

We did a preliminary review of TokenVaultV2, which fixes many issues with the current TokenVault. Instead of storing assets in a single contract, the new vault leverages the `CREATE2` opcode to create per-NFT vaults. This greatly simplifies the internal accounting, reducing the attack surface. Overall, this represents a major step towards a stronger security posture. The new vaults are not yet production ready. Some features, such as migrations, are implemented but not yet fully complete and functional.

In the `fnft-migrations` branch, we noted that the function `_beforeTokenTransfer` calls back to `IOutputReceiverV4`. Interestingly, it does not call back on `mint` operations due to the check `from` $\neq$ `0`, whereas it does call back on `burn` operations. Additionally, 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, these inconsistencies impede composability. We therefore recommend documenting these side-effects or simplifying them.

In MetadataHandler.sol, several setter functions reponsible for setting the rendering and Token URI (e.g., `setTokenURI` and `setRenderTokenURI`) lack access controls. That being said, these functions are not crucial to business logic of the system, except for the frontend.

Test suite maturity

We were unable to verify the test suite in its complete capacity, as there were some errors which prevented a successful run. Notwithstanding that, it seems there has been an extensive effort to test functions. Previous bugs and vulnerabilities are exercised in regression tests, such as a PoC for a past exploit.

Re-entrancy points

User control of execution mostly stems from the fact that users exercise 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 (`fnft-migration` branch)

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

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

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