Security Audit Report for Revest Contracts

Date: June 3, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.06.03 10:55:43 +08'00'

Contents

Report Manifest

Item	Description
Client	Revest
Target	Revest Contracts

Version History

Version	Date	Description
1.0	June 03, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repository that has been audited includes Revest [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project	Branch	Commit SHA
Version 1		
Revest	feature/fnft-migrator	6b06a1350aae57f6dd70afbed27e7e894b84c13e
Revest	feature/token-vault-overhaul	8c8379eae8fde55dc724997f5791aeb9f4ed4c27
Version 2		
Revest	feature/fnft-migrator	302c55f67989595a35c1e3cf3adcf05d77ea09e2
Revest	feature/token-vault-overhaul	eb22042402db995aba43423cdc634d8b31335f32
Revest	bug/uniswap-twap	6f08e90ac999818e860919714b465ea77bdc3143

Note that, we did **NOT** audit the ’demo/unimplemented’ folder and the ’staking’ folder in the repository.

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/Revest-Finance/Revest

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

		Likelihood	
		High	*Low*
Impact	*High*	High	Medium
	Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **four** potential issues. We have **ten** recommendations.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 3
- Recommendations: 10

ID	Severity	Description	Category	Status
1	Low	The `balanceOf` of the *TokenVault* contract is at risk of being manipulated	DeFi Security	Fixed
2	Low	The potential mistake in the library *RevestHelper*	DeFi Security	Fixed
3	Low	The potential mistake in the *Staking* contract	DeFi Security	Fixed
4	-	Check the variable `fnft.split`	Recommendation	Fixed
5	-	Merge the two functions, i.e., `manualMapRVSTBasic` and `manualMapWETHBasic`	Recommendation	Confirmed
6	-	Remove the useless code I	Recommendation	Fixed
7	-	Remove the useless code II	Recommendation	Fixed
8	-	Make the *MetadataHandler* contract read-only	Recommendation	Fixed
9	-	Address the concern of the centralization design	Recommendation	Confirmed
10	-	Update the total allocation point in the `manualMap*` functions	Recommendation	Confirmed
11	-	Remove the useless code III	Recommendation	Fixed
12	-	Make the `splitFNFT` function compatible with the new design	Recommendation	Fixed
13	-	Add a check in the `batchMint` function	Recommendation	Confirmed
14	-	Make sure the *Staking* contract is on the fee white list of the *Revest* contract	Notes	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 The `balanceOf` of the *TokenVault* contract is at risk of being manipulated

Severity Low

Status Fixed in the *TokenVaultV2* contract of `Version 1`'s feature/token-vault-overhaul branch

Introduced by the *TokenVault* contract of `Version 1`

Description The *TokenVault* contract has a mechanism to handle the rebase (or deflation token). First, it records a global variable `tracker` that is updated before each deposit, which is shown in the below formula.

We use the `incomingDeposit` variable to represent the amount that will be locked. [1]

$$
\begin{aligned}
&lastBal = tracker.lastBalance \\
&tracker.lastMul = \begin{cases} 1e18 & lastBal == 0 \\ 1e18 * \frac{asset.balanceOf(this)}{lastBal} & lastBal > 0 \end{cases} \\
&tracker.lastBalance = asset.balanceOf(this) + incomingDeposit
\end{aligned}
\tag{2.1}
$$

Second, it also records the current `tracker.lastMul` for each lock.

$$
\begin{aligned}
&fnfts[fnftId].amount = incomingDeposit \\
&fnfts[fnftId].depositMul = tracker.lastMul
\end{aligned}
\tag{2.2}
$$

Finally, the asset that can be withdrawn is as follows:

$$
withdrawAmount = fnfts[fnftId].amount * \frac{tracker.lastMul}{fnfts[fnftId].depositMul}
\tag{2.3}
$$

If the asset is not a deflation token or rebase token, then the `tracker.lastMul` is expected as a constant variable: $1e18$, and the `withdrawAmount` is equal to the `incomingDeposit`. Therefore, the mechanism does not affect the bookkeeping of normal assets. However, the `tracker.lastMul` can be manipulated by donating assets, because it's calculation depends on the `asset.balanceOf(this)`.

We then provide an attack scenario that can keep the user's assets in the contract forever. To illustration, we assume the asset is WETH, and the `fnftId` is 0. The attack consists of three steps:

1. An attacker first launches a malicious deposit that lock $1wei$ WETH in the contract, then the contract's states are shown in below:

$$
\begin{aligned}
&tracker.lastMul = 1e18 \\
&tracker.lastBalance = 0 + 1 = 1 \\
&fnfts[0].amount = 1 \\
&fnfts[0].depositMul = 1e18 \\
&WETH.balanceOf(this) = 1
\end{aligned}
\tag{2.4}
$$

2. The attacker donates $1ether$ WETH to the contract.

$$
WETH.balanceOf(this) = 1 + 1e18
\tag{2.5}
$$

3. A victim locks $5ether$ WETH in the contract.

$$
\begin{aligned}
&tracker.lastMul = 1e18(1e18 + 1) \\
&tracker.lastBalance = 1e18 + 1 \\
&fnfts[1].amount = 1e18 \\
&fnfts[1].depositMul = 1e18(1e18 + 1)
\end{aligned}
\tag{2.6}
$$

4. The victim withdraws his deposits after unlocking them.

$$
\begin{aligned}
&tracker.lastMul = 1e18 \\
&withdrawAmount = 1e18 * \frac{1e18}{1e18(1e18 + 1)} = 0
\end{aligned}
\tag{2.7}
$$

The victim unlocks his deposits but get nothing due to the `withdrawAmount` is zero.

[1] In order to simplify the description, we consider only one asset and one quantity of FNFT.

In addition, since the attacker has no profits but costs during the attack, we mark this issue as a low risk one.

Impact This issue may incur an attack that prevents users from withdrawing their assets.

Suggestion Use a variable `reserve` to record the amount of assets that locked in the contract rather than use `balanceOf`.

2.1.2 The potential mistake in the library *RevestHelper*

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `getLockType` returns the human-readable string of a specified lock type. However, there is a mistake that causes the function returns either "Time" or "".

```
function getLockType(IRevest.LockType lock) private pure returns (string memory lockType) {
    if(lock == IRevest.LockType.TimeLock) {
        lockType = "Time";
    }
    if(lock == IRevest.LockType.TimeLock) {
        lockType = "Value";
    }
    if(lock == IRevest.LockType.TimeLock) {
        lockType = "Address";
    }
}
```

Listing 2.1: utils/RevestHelper.sol

There is no contract to call the function in this project, but considering it may bring error information to the front end, we still mark it as a low risk issue.

Impact The function `getLockType` can not return the correct string.

Suggestion Repair the mistake.

2.1.3 The potential mistake in the *Staking* contract

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description As shown in the below code snippet, the `_depositAdditionalToStake` function is used to claim the user's rewards and reset the user's allocation points before supplying additional assets in an existing lock, and this function is invoked by the *Revest* contract by design. Therefore, the second parameter of the invocation `claimRewards` (in line 228) should be `caller` rather than `_msgSender()`.

```
218    function _depositAdditionalToStake(uint fnftId, uint amount, address caller) private {
219        //Prevent unauthorized access
220        require(IFNFTHandler(getRegistry().getRevestFNFT()).getBalance(caller, fnftId) == 1, 'E061'
               );
```

```
        require(fnftId > previousStakingIDCutoff, 'E080');
        uint time = stakingConfigs[fnftId].timePeriod;
        require(time > 0, 'E078');
        address asset = ITokenVault(getRegistry().getTokenVault()).getFNFT(fnftId).asset;
        require(asset == revestAddress || asset == lpAddress, 'E079');

        //Claim rewards owed
        IRewardsHandler(rewardsHandlerAddress).claimRewards(fnftId, _msgSender());

        //Write new, extended unlock date
        stakingConfigs[fnftId].dateLockedFrom = block.timestamp;
        stakingConfigs[fnftId].amount = stakingConfigs[fnftId].amount + amount;
        //Retreive current allocation points - WETH and RVST implicitly have identical alloc points
        uint oldAllocPoints = IRewardsHandler(rewardsHandlerAddress).getAllocPoint(fnftId,
            revestAddress, asset == revestAddress);
        uint allocPoints = amount * getInterestRate(time) + oldAllocPoints;

        _updateShares(asset, fnftId, allocPoints);

        emit DepositERC20OutputReceiver(_msgSender(), asset, amount, fnftId, '');
    }
```

Listing 2.2: demo/output_receivers/Staking.sol

Impact This issue will cause some users not to receive the reward token they deserve.

Suggestion Repair the mistake.

2.2 Additional Recommendation

2.2.1 Check the variable `fnft.split`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The variable `fnft.split` seems to be used to record the times the specified lock can be split. As shown in below code, the variable `newFNFT.split` is reduced by one for each split, but there is no code to check if it is zero. Normally, if so, the split should be reverted.

```
    for(uint i = 0; i < proportions.length; i++) {
        runningTotal += proportions[i];
        uint amount = fnft.depositAmount * proportions[i] / denominator;
        IRevest.FNFTConfig memory newFNFT = cloneFNFTConfig(fnft);
        newFNFT.depositAmount = amount;
        newFNFT.split -= 1;
        mapFNFTToToken(newFNFTIds[i], newFNFT);
        emit CreateFNFT(newFNFTIds[i], _msgSender());
    }
```

Listing 2.3: TokenVaultV2.sol:splitFNFT

Since the function is disabled in `Version 1`, we do not mark it as an issue.

Impact NA.

Suggestion Add a check `require(newFNFT.split >= 0; "XXX" )` in the function `splitFNFT`.

2.2.2 Merge the two functions, i.e., `manualMapRVSTBasic` and `manualMapWETHBasic`

Status Confirmed

Introduced by Version 1

Description There is an implicit assumption in the contracts *RewardsHandler* and *Staking*: the `allocPoint` is the same for `wethBasic` and `rvstBasic`. The function `manualMapRVSTBasic` updates the `rvstBasic`, and the function `manualMapWETHBasic` updates the `wethBasic`. It's better to combine the two functions into one.

Impact NA.

Suggestion Combine the two functions: `manualMapRVSTBasic` and `manualMapWETHBasic`, and merge the two functions: `manualMapRVSTLP` and `manualMapWethLP`.

Feedback from the Project We're considering removing these functions permanently.

2.2.3 Remove the useless code I

Status Fixed in Version 2

Introduced by Version 1

Description There are a little useless code:

```
function rewardsOwed(address token, UserBalance memory lpBalance, UserBalance memory basicBalance) internal view returns (uint) {
    uint globalBalance = IERC20(token).balanceOf(address(this));
    uint lpRewards = (getLPGlobalMul(token) - lpBalance.lastMul) * lpBalance.allocPoint;
    uint basicRewards = (getBasicGlobalMul(token) - basicBalance.lastMul) * basicBalance.allocPoint;
    uint tokenAmount = (lpRewards + basicRewards) / PRECISION;
    return tokenAmount;
}
```

Listing 2.4: RewardsHandler.sol

The `globalBalance` variable in the function `rewardsOwed` is not used.

```
constructor(address provider, address weth, address uni_v3, uint24 uniFee) RevestAccessControl(provider) {
    WETH = weth;
    UNIV3_FACTORY = uni_v3;
    UNIV3_FEE = uniFee;
}
```

Listing 2.5: oracles/UniswapV3CronjeSun.sol

The three variables: `WETH`, `UNIV3_FACTORY`, and `UNIV3_FEE` are initialized but not used.

Impact NA.

Suggestion Remove the useless code.

2.2.4 Remove the useless code II

Status Fixed in Version 2's bug/uniswap-twap branch

Introduced by Version 1

Description There are a little useless code:

```
67    if(block.timestamp <= twap.timestampLatest + MIN_UPDATE) {
68        // Wait until safe update period has passed to update TWAP
69        return false;
70    }
71    IUniswapV2Pair pair = IUniswapV2Pair(twap.pairAddress);
72    bool inverted = pair.token1() == asset;
73    uint cumLast = inverted ? pair.price1CumulativeLast() : pair.price0CumulativeLast();
74
75    uint lastTimeTwapUpdated = twap.timestampLatest;
76    uint lastPrice = twap.lastUpdateCumulativePrice;
77
78    (, , uint lastTime) = pair.getReserves();
79
80    if(cumLast - lastPrice <= 0 || lastTime - lastTimeTwapUpdated <= 0) {
81        // There has been no value on the Uniswap pair since the last update
82        // Attempt to force the uni pair to sync
83        pair.sync();
84        // Reset variables
85        cumLast = inverted ? pair.price1CumulativeLast() : pair.price0CumulativeLast();
86        (, , lastTime) = pair.getReserves();
87        if(cumLast - lastPrice <= 0 || lastTime - lastTimeTwapUpdated <= 0) {
88            // If this has failed, we must return false
89            return false;
90        }
91    }
92
93    if(!twap.initialized) {
94        if(twap.timestampLatest == 0) {
95            twap.timestampLatest = lastTime;
96            twap.pairAddress = IUniswapV2Factory(uniswap).getPair(asset, compareTo);
97        }
```

Listing 2.6: oracles/UniswapTwapOracleDispath.sol:updateOracle

Since the check in line 67, the code from line 94 to 97 is dead code.

Impact NA.

Suggestion Remove the useless code.

2.2.5 Make the *MetadataHandler* contract read-only

Status Fixed in Version 2

Introduced by Version 1

Description According to the document, the *MetadataHandler* contract is a read only contract. However, the below two functions that change the contract's states can be invoked anyone.

```
function setTokenURI(uint fnftId, string memory _uri) external override {
    uri = _uri;
}
```

Listing 2.7: MetadataHandler.sol

```
function setRenderTokenURI(
    uint tokenID,
    string memory baseRenderURI
) external override {
    renderURI = baseRenderURI;
}
```

Listing 2.8: MetadataHandler.sol

Impact NA.

Suggestion Add the `onlyOwner` modifier to the two functions: `setTokenURI` and `setRenderTokenURI`.

2.2.6 Address the concern of the centralization design

Status Confirmed

Introduced by `Version 1`

Description NA.

Impact NA.

Suggestion There are a few critical actions that can be invoked by only owners. Especially, the contract *RevestPublicSaleBatch* is almost a centralized contract that sells assets off-chain. Considering the risk of leakage of these authorized EoA private keys, we recommend the project adopt a decentralized method to manage authority (e.g., DAO contract), or leverage a secure private key solution (e.g., multi-signed wallet, and TEE based security key management) to manage the private keys of authorized EOAs.

Feedback from the Project We're using the multi-signed wallet and gradually transitioning to a DAO.

2.2.7 Update the total allocation point in the `manualMap*` functions

Status Confirmed

Introduced by `Version 1`.

Description There is an implicit condition that the total allocation point is sum of users' allocation points in the *RewardsHandler* contract. However, there are a few `manualMap*` functions that manually set the user's allocation point but not update the total allocation point accordingly. The below code snippet takes the `manualMapRVSTBasic` function as an example to show that.

```
function manualMapRVSTBasic(
    uint[] memory fnfts,
    uint[] memory allocPoints
) external onlyOwner {
    for(uint i = 0; i < fnfts.length; i++) {
        UserBalance storage userBal = rvstBasicBalances[fnfts[i]];
        userBal.allocPoint = allocPoints[i];
```

```
            userBal.lastMul = rvstBasicGlobalMul;
        }
    }
```

Listing 2.9: RewardsHandler.sol

Impact NA.

Suggestion Update the total allocation point in the functions: `manualMapRVSTBasic` and `manualMapRVSTLP` of the `manualMapRVSTBasic` contract, `manualMapRVSTLP`, `manualMapWethBasic`, and `manualMapWethLP` of the *RewardsHandler* contract.

Feedback from the Project Future versions of this contract wouldn't contain these methods to begin with, and their inclusion was only out of necessity during an initial failure in our staking system.

2.2.8 Remove the useless code III

Status Fixed in `Version 2`

Introduced by `Version 1`'s feature/token-vault-overhaul branch.

Description

```
285     function _withdrawFNFT(uint fnftId, uint quantity) private {
286         address fnftHandler = addressesProvider.getRevestFNFT();
287 
288         // Check if this many FNFTs exist in the first place for the given ID
289         require(quantity > 0, "E003");
290         // Burn the FNFTs being exchanged
291         IFNFTHandler(fnftHandler).burn(_msgSender(), fnftId, quantity);
292         require(getLockManager().unlockFNFT(fnftId, _msgSender()), 'E082');
293         address vault = addressesProvider.getTokenVault();
294 
295         // This code snippet auto-patches old output receivers to new ones
296         // Allows us to solve the problem of migrating token vaults for output receivers
297         IRevest.FNFTConfig memory config = ITokenVault(vault).getFNFT(fnftId);
298         // console.log("Vault: %s", vault);
299 
300         ITokenVault(vault).withdrawToken(fnftId, quantity, _msgSender());
301         emit FNFTWithdrawn(_msgSender(), fnftId, quantity);
302     }
```

Listing 2.10: Revest.sol

The code in line 297 is not used.

Impact NA.

Suggestion Remove the useless code.

2.2.9 Make the `splitFNFT` function compatible with the new design

Status Fixed in `Version 2`

Introduced by `Version 1`'s feature/token-vault-overhaul branch

Description The TokenVaultV2 contract has already uses the *RevestSmartWallet* contract to handle the rebase or deflationary token. The new design almost disabled the field depositAmount, but the splitFNFT function still uses it to record amount.

```
function splitFNFT(
    uint fnftId,
    uint[] memory newFNFTIds,
    uint[] memory proportions,
    uint quantity
) external override onlyRevestController {
    IRevest.FNFTConfig storage fnft = fnfts[fnftId];
    // Burn the original FNFT but keep its lock

    // Create new FNFTs with the same config, only thing changed is the depositAmount
    // proportions should add up to 1000
    uint denominator = 1000;
    uint runningTotal = 0;
    for(uint i = 0; i < proportions.length; i++) {
        runningTotal += proportions[i];
        uint amount = fnft.depositAmount * proportions[i] / denominator;
        IRevest.FNFTConfig memory newFNFT = cloneFNFTConfig(fnft);
        newFNFT.depositAmount = amount;
        newFNFT.split -= 1;
        mapFNFTToToken(newFNFTIds[i], newFNFT);
        emit CreateFNFT(newFNFTIds[i], _msgSender());
    }
    // This is really a precondition but more efficient to place here
    require(runningTotal == denominator, 'E054');
    if(quantity == getFNFTHandler().getSupply(fnftId)) {
        // We should also burn it
        removeFNFT(fnftId);
    }
    emit RedeemFNFT(fnftId, _msgSender());
}
```

Listing 2.11: TokenVaultV2.sol

Since the function is disabled in Version 1, we do not mark it as an issue.

Impact NA.

Suggestion Update the splitFNFT function to compatible with the new design.

2.2.10 Add a check in the batchMint function

Status Confirmed

Introduced by Version 1's feature/fnft-migration branch

Description The below function batchMint is used to migrate the existing fnfts from the old *FNFTHandler* contract to the new one. Specifically, the project manually passes the old one's state to the batchMint function that then invokes the new one's mintBatchRec function to migrate the state.

```
function batchMint(address[][] memory recipients, uint[][] memory balances, uint[] memory ids,
     uint[] memory supplies) external {
        require(msg.sender == OWNER, "!AUTH");
        for(uint i = 0; i < recipients.length; i++) {
            address[] memory recips = recipients[i];
            uint[] memory bals = balances[i];
            uint id = ids[i];
            IFNFTHandler(FNFT_HANDLER).mintBatchRec(recips, bals, id, supplies[i], ’0x0’);
        }
    }
```

Listing 2.12: utils/FNFTHandlerMigrator.sol

Note that, the `mintBatchRec` function does not verify if the `newSupply` is equal to the sum of `quantities`, as shown in the below code snippet. Therefore, the verification should be performed by the function `batchMint`.

```
    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);
        }
    }
```

Listing 2.13: FNFTHandler.sol

Impact NA.

Suggestion Add a check in the `batchMint` function of the *FNFTHandlerMigrator* contract to make sure that the sum of `bals[i]` is equal to the `supplies[i]`.

Feedback from the Project This function will only ever be called with admin-originating code, and the verification can be performed with JS. This saves gas.

2.3 Notes

2.3.1 Make sure the *Staking* contract is on the fee white list of the *Revest* contract

Status Confirmed

Introduced by `Version 1`

Description For the caller who is not on the white list, the *Revest* contract charges Ether fees for each lock.

```
    if(!whitelisted[_msgSender()]) {
        if(flatWeiFee > 0) {
            require(weiValue >= flatWeiFee, "E005");
```

Listing 2.14: Revest.sol:doMint

```
    uint fnftId = IRevest(revest).mintAddressLock(address(this), ’’, recipients, quantities, fnftConfig);
```

Listing 2.15: Staking.sol:_staking

As shown in above code, the *Staking* contract does not pay the fee to the *Revest* via `msg.value`. Therefore, we recommend that you make sure the *Staking* contract is inside the fee whitelist of the *Revest* contract. Otherwise, the invocation to the function `_staking` will be reverted.

Feedback from the Project Acknowledge and agreed upon.