Perfect Abstractions

Description	Universe XYZ Marketplace Audit
Copyright	Copyright © 2022 - Perfect Abstractions LLC

Table of Contents

1 Universe XYZ Marketplace Audit

Perfect Abstractions conducted a smart contract audit of Universe XYZ's Universe Marketplace from 21 February 2022 to 4 March 2022.

The git commit hash used for the audit is `e88f77b124d513ee859ad56a106ecb88e131f7a0`.

Auditors:

- Nick Mudge

1.0.1 Objectives

1. Find bugs, inefficiencies and security vulnerabilities in the code base.
2. Make recommendations concerning bugs, inefficiencies and security vulnerabilities found as well as other recommendations that may improve the code base.

1.0.2 Scope

The following files were audited:

- contracts/UniverseMarketplaceCore.sol
- contracts/UniverseMarketplace.sol
- contracts/ERC721FloorBidMatcher.sol
- contracts/transfer-proxy/TransferProxy.sol
- contracts/transfer-proxy/ERC721LazyMintTransferProxy.sol
- contracts/transfer-proxy/ERC20TransferProxy.sol
- contracts/transfer-manager/SimpleTransferManager.sol
- contracts/transfer-manager/UniverseTransferManager.sol
- contracts/transfer-executor/TransferExecutor.sol
- contracts/royalties/ERC2981Royalties.sol
- contracts/royalties/HasSecondarySaleFees.sol
- contracts/royalties/RoyaltiesRegistry.sol
- contracts/order-validator/OrderValidator.sol
- contracts/operator/OperatorRole.sol
- contracts/lib/BpLibrary.sol
- contracts/lib/LibAsset.sol
- contracts/lib/LibERC1155LazyMint.sol
- contracts/lib/LibERC721LazyMint.sol

- contracts/lib/LibFeeSide.sol
- contracts/lib/LibFill.sol
- contracts/lib/LibMath.sol
- contracts/lib/LibOrder.sol
- contracts/lib/LibOrderData.sol
- contracts/lib/LibPart.sol
- contracts/lib/LibSignature.sol
- contracts/lib/LibTransfer.sol
- contracts/asset-matcher/AssetMatcher.sol

I. Medium Risk

2 Upgrade signature validation

Medium Risk

Fixed

Fixed according to the recommendation.

This is the code in OrderValidator.sol that validates signatures:

```
bytes32 hash = LibOrder.hash(order);
if (_hashTypedDataV4(hash).recover(signature) != order.maker) {
    if (order.maker.isContract()) {
        require(
            ERC1271(order.maker).isValidSignature(_hashTypedDataV4(hash), signature) ==
MAGICVALUE,
            "contract order signature verification error"
        );
    } else {
        revert("order signature verification error");
    }
}
```

2.0.1 `recover` reverts internally so `isValidSignature` can't be called

A problem with this code is that the `recover()` function reverts internally if it determines a signature to be invalid.

It is possible for `ERC1271(order.maker).isValidSignature` to validate a signature differently. What may be valid for `ERC1271(order.maker).isValidSignature` may not be valid for `recover`.

But `ERC1271(order.maker).isValidSignature` will never get a chance to validate signatures if `recover` reverts.

2.0.2 `recover` does not support short signatures

The `recover` function comes from `LibSignature.sol` which is coped from ECDSAUpgradeable.sol#release-v4.3, which is an older version that does not support short signatures.

Current versions of `recover` in `ECDSAUpgradeable.sol` from OpenZeppelin support short signatures.

2.1 Recommendation

Delete `LibSignature.sol` and use the `isValidSignatureNow` function from SignatureCheckerUpgradeable.sol instead.

The `isValidSignatureNow` function combines the `recover` and `isValidSignature` function calls into a single function call in a way that solves Item 1 and Item 2 mentioned above.

The code could look like this:

At the top of the `OrderValidate.sol` file import `SignatureCheckerUpgradeable`:

```
import "@openzeppelin/contracts-upgradeable/utils/cryptography/SignatureCheckerUpgradeable.sol";
```

Then use it:

```
bytes32 hash = LibOrder.hash(order);
require(SignatureCheckerUpgradeable.isValidSignatureNow(order.maker, _hashTypedDataV4(hash),
signature), "order signature verification error");
```

II. Low Risk

3 ERC721FloorBidMatcher blockchain reorg vulnerability

Low Risk

A blockchain reorganization occurs when accepted blocks of transactions are thrown out and replaced with other blocks of transactions when a longer chain of truth is found. More information blockchain reorgs here: https://medium.com/blockvigil/how-we-deal-with-chain-reorganization-at-ethvigil-5a8c06859c7

Blockchain reorgs can cause vulnerabilities in smart contracts (that don't account for it) when users or software see and act on information in a blockchain that is later thrown out because of a blockchain reorganization.

A blockchain reorg vulnerability exists in `ERC721FloorBidMatcher` due to the following circumstances:

1. The `createBuyOrder` functions create order IDs sequentially.
2. The `matchBuyOrder` function uses `uint256 orderId` to identify orders.

Here is what can happen:

1. User A (or some bot or other software) sees a new order of NFTs from address X at the price of 10 USDC. The order ID is 112.
2. User A submits a transaction to execute the `matchBuyOrder` function using order ID 112 to sell his NFTs at that price (10 USDC).
3. While User A's transaction is pending a block reorganization occurs that throws out the transaction that created order 112 and replaces it with a different transaction that creates a different order that is also order 112 (because order ids are sequentially created). This different order has a different price, let's say the price is 1 USDC instead of 10 USDC.
4. User A's transaction executes the new/different order 112 and so the user sells his NFTs at a different price than he/she meant to.

3.1 Recommendation

A way to prevent this is to generate the order id from the important information of the order plus an incrementing value like `block.number`. For example order ids could be generated like this:

```
uint256 orderId = keccak256(abi.encodePacked(_msgSender(), erc721TokenAddress,
paymentTokenAddress, tokenPrice, block.number))
```

Another way to prevent this is by using additional parameters in the `matchBuyOrder` function to identify the order. For example `matchBuyOrder` could have these parameters:

```
function matchBuyOrder(
    uint256 orderId,
    uint256[] calldata tokenIds,
    address erc721TokenAddress,
    address paymentTokenAddress,
    uint256 tokenPrice
)
```

And those parameters can be compared to the order's data to ensure the right order is executed.

Note: While I have seen this problem actually occur on Polygon I have not seen it occur on Ethereum. So I don't know how likely this issue could happen. One thing that can be looked at is how often chain reorgs happen. Etherscan shows chain reorg info here: https://etherscan.io/blocks_forked

4 Handling ERC20 token transfers that don't have a return value

Low Risk

Fixed

Fixed according to the recommendation.

The ERC20TransferProxy contract transfers ERC20 tokens however it expects that the transfers return a bool value. This is correct and according to the ERC20 standard. However some non-standard ERC20 tokens on Ethereum do not return a value. You still might want to support these ERC20 tokens, such as USDT.

More information about this here: https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca

4.1 Recommendation

A way to handle this problem is to use code that correctly handles transfer return values and transfers that don't have return values. Here is a link to a library that does this: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

5 Add nonReentrant function to the matchOrders function

Low Risk

Done

Done according to recommendation.

Due to the complexity and multiple state changing external function calls, to ensure no possible reentrance vulnerability add the `nonReentrant` modifier to the `matchOrders` function in the `UniverseMarketplaceCore` contract.

III. Informational

6 Order of fill orders

Informational

Decision

Decided to keep the initial implementation.

It is unclear how orders are supposed to filled because there is no clear specification or documentation that states the different order scenarios and what should happen.

From reading the code and rarible documentation here's how orders could be filled:

1. Either all of the left order is filled or all the right order is filled, or both are totally filled.
2. The left order is filled if the right order take value is greater than the left order make value, otherwise the right order is filled.
3. If fulfilling an order will cause an order (left or right) to be exchanged at less than its exchange rate then the transaction will revert.

However the above three rules are not completely followed because the `fillRight` function returns `makerValue` instead of `rightTakeValue`. The rarible protocol has an unmerged closed pull request about this same thing here: https://github.com/rarible/protocol-contracts/pull/127

Specifically rule number 2 is violated in this scenario:

```
Left order:
make: 100X
take: 50Y

Right order:
make: 51Y
take: 100X
```

The left order gets fully filled instead of the right order.

Perhaps `fillRight` can be kept how it currently is if that is preferred. It seems rarible protocol is keeping it how it is. A good thing with how it currently works is that if someone makes a mistake and uses a bad/wrong exchange rate for the right order the exchange rate of the left order will be used.

I suggest writing tests for the fillOrder function that covers every scenario possible with filling orders, to be sure that orders are filled in every way expected in the future.

The rarible protocol has an example of testing different fill order scenarios here: https://github.com/rarible/protocol-contracts/blob/master/exchange-v2/test/v2/LibFill.test.js

7 Case where no NftRoyalties value is returned

Informational

Fixed

Code was changed to retrieve the nftRoyalties value.

Concerning the `getRoyalties` function in the `RoyaltiesRegistry` contract:

If `royaltiesSetCollection` is cached but `royaltiesSetNFT` is not cached then there is no check for `royaltiesSetNFT` and it is set to no royalties.

This may be the intended behavior.

The code:

```
if (royaltiesSetNFT.initialized) {
    nftRoyalties = royaltiesSetNFT.royalties;
}

if (royaltiesSetNFT.initialized || royaltiesSetCollection.initialized ) {
    return (nftRoyalties, collectionRoyalties);
}
```

8 No indexed fields in ERC721FloorBidMatcher events

Informational

None of the events in ERC721FloorBidMatcher have indexed fields.

You may want to consider making some of the fields indexed so that user interfaces or other software can more easily find/search certain event information. However it is unknown if this is necessary for this project or not.

9 ERC721FloorBidMatcher.sol gas optimizations

Informational

9.0.1 1. Packing ERC721FloorBidOrder Struct

The `numberOfTokens` and `endTime` variables in the ERC721FloorBidOrder struct can use smaller data types. Doing this will cause less gas cost to read and write to the `ERC721FloorBidOrder` struct as fewer storage slots will be used.

I suggest the struct to be changed to the following:

```
    struct ERC721FloorBidOrder {
        address erc721TokenAddress;
        uint16 numberOfTokens;
        uint256[] erc721TokenIdsSold;
        uint256 tokenPrice;
        address paymentTokenAddress;
        uint256 amount;
        uint32 endTime;
        address creator;
        ERC721FloorBidMatcher.OrderStatus orderStatus;
    }
```

9.0.2 2. NftTransferProxy and ERC20TransferProxy contracts increase gas costs

The transfer proxy contracts add gas overhead to transferring tokens.

In the `createBuyOrder` an ERC20 transfer proxy adds the following overhead:

1. The gas cost to call the `approve` external function which involves state variable reads and writes.
2. The gas cost to call the `erc20safeTransferFrom` external function on the ERC20 proxy.
3. Calling `erc20safeTransferFrom` on the ERC20 proxy causes the proxy to read a state variable and make a second external function call using delegatecall -- this is mechanics and gas overhead of the proxy implementation.
4. The `erc20safeTransferFrom` function itself does a state read for operator authorization.

Direct ERC20 and ERC721 transfers are gas expensive. Using ERC20 transfer proxies increases the cost further. Since the Berlin hardfork last year the gas cost for an external function call increased from 700 gas to 2600 gas, and storage slot reads increased from 800 to 2100. But this is for the first access to an external contract or storage reads. Subsequent external calls to the same contracts and reads to the same storage slots cost 100 gas. So it makes sense to optimize away from external function calls and storage reads.

The `NftTransferProxy` contract is called in a loop in the `matchBuyOrder` function.

The `createBuyOrder`, `matchBuyOrder`, `cancelOrder`, `withdrawFundsFromExpiredOrder` functions are using transfer proxies.

The transfer proxy function calls can be replaced with direct transfer calls which will cost less gas for users.

Using a transfer proxy for NFTs has a benefit that it makes it easy to add support for non-standard ERC721 contracts and new NFT standards in the future. However `ERC721FloorBidMatcher` is an upgradeable contract and could be upgraded in the future to support other NFTs transfers directly.

One idea is to do a gas cost analysis of `ERC721FloorBidMatcher.sol`. Could compare the gas costs to using transfer proxies versus not using them, and find out or determine what is acceptable. Keep in mind that a project generally has no or very little control of gas prices on Ethereum. Gas prices may increase or decrease in the future.

Related issue here: Gas Optimize Transfers in TransferExecutor.sol #5

9.0.3 3. Not Necessary to Store `amount` variable in `ERC721FloorBidOrder` struct

The `numberOfTokens` variable times the `tokenPrice` variable in the `ERC721FloorBidOrder` struct provide the same value as the `amount` variable, so the `amount` variable does not need to exist in the `ERC721FloorBidOrder` struct.

Removing the `amount` variable from the `ERC721FloorBidOrder` struct saves 20,000 gas in the `createBuyOrder` functions and 5,000 gas in the `matchBuyOrder` function.

I also suggest using the `uint256 tokenPrice` parameter in the `createBuyOrder` function instead of the `uint256 amount` parameter. Because the `uint256 amount` parameter can have a small rounding error when calculating `uint256 tokenPrice`.

9.0.4 4. State Variables Read Multiple Times

The first time a state variable is read it costs 2100 gas. Subsequent reads of the same state variable in the same transaction cost 100 gas. Reading a local variable costs no or very little gas.

If a state variable will be used multiple times in the same function it will cost less gas if the state variable value is assigned to a local variable and the local variable is used.

In the `matchBuyOrder` function the `order.erc721TokenAddress` state variable is read twice within a loop.

I recommend assigning state variable values to local variables when they are used more than once in the same function, or when the same values are used in loops. This change can apply to the `matchBuyOrder` function and other functions that are reading the same state variables more than once in the same function or using the same state variables in loops.

9.0.5 5. Consider removing `erc721TokenIdsSold` from the ERC721FloorBidOrder struct

The `matchBuyOrder` function adds each tokenId of an NFT that is sold to the `order.erc721TokenIdsSold` array. This adds a gas cost of 20,000 gas for each NFT that is sold. So if `matchBuyOrder` sold 20 NFTs this adds a gas cost of 20,000 * 20 which is 400,000 gas. At today's low price of gas of about 85 gwei that is $88.66 USD.

If possible I suggest removing the `erc721TokenIdsSold` variable from the ERC721FloorBidOrder struct to save gas. Other ways can be used to show what NFTs are sold. Block explorers show ERC721 transfers for both parties. The `LogMatchBuyOrder` event that is emitted shows what tokenIds were sold.

I suggest removal of `erc721TokenIdsSold` because it adds significant gas and is not necessary for the operation of the exchange smart contract logic. It is only records data that can be retrieved with the `getSoldTokensFromOrder(uint256 orderId)` function.

9.0.6 6. nonReentrant Modifier

If the Checks-Effects-Interactions Pattern is used then some of the functions currently using the nonReentrant modifier don't need to use it. The `nonReentrant` modifier adds about 2244 gas to a function according to my testing.

Examples of functions that could use the Checks-Effects-Interactions Pattern pattern instead of the `nonReentrant` modifier: `createBuyOrder`, `createBuyOrderETH`, `cancelOrder`, `withdrawFundsFromExpiredOrder`.

10 No indexed fields on Cancel and Match events

Informational

Done

Indexes were added to `Cancel` and `Match` events.

There are no indexed fields in the `Cancel` and `Match` events.

It may be useful for user interfaces or other software systems to use indexes on some of the event fields to quickly find certain events, like all the orders from a particular address, or getting information about a specific order, etc:

```
event Cancel(bytes32 hash, address maker, LibAsset.AssetType makeAssetType, LibAsset.AssetType
takeAssetType);
event Match(bytes32 leftHash, bytes32 rightHash, address leftMaker, address rightMaker, uint
newLeftFill, uint newRightFill, LibAsset.AssetType leftAsset, LibAsset.AssetType rightAsset);
```

11 Modify EIP712 implementation to use constants for _HASHED_NAME and _HASHED_VERSION

Informational

Done

Recommended changes were made in a new file.

The `draft-EIP712Upgradeable.sol` implementation being used from OpenZeppelin uses `bytes32 private _HASHED_NAME` and `bytes32 private _HASHED_VERSION` state variables which is gas inefficient because these could instead be constants.

Reading a state variable costs 2100 gas. Reading a constant costs nothing since the compiler replaces constants with literal values during compilation.

The `_hashTypedDataV4` function from `draft-EIP712Upgradeable.sol` reads the state variables `_HASHED_NAME` and `_HASHED_VERSION`.

`_hashTypedDataV4` is used by the `matchOrders` function in `UniverseMarketplaceCore` to verify signatures.

11.0.1 Recommendation

I recommend copying the `draft-EIP712Upgradeable.sol` implementation to the Universe-Marketplace repo and making the following changes:

Change this in `draft-EIP712Upgradeable.sol`:

```
bytes32 private _HASHED_NAME;
bytes32 private _HASHED_VERSION;
```

To this:

```
bytes32 private constant _HASHED_NAME = keccak256("Exchange");
bytes32 private constant _HASHED_VERSION = keccak256("2");
```

And delete these functions in `draft-EIP712Upgradeable.sol` because they aren't needed:

```
    function __EIP712_init(string memory name, string memory version) internal initializer {
        __EIP712_init_unchained(name, version);
    }

    function __EIP712_init_unchained(string memory name, string memory version) internal
initializer {
        bytes32 hashedName = keccak256(bytes(name));
        bytes32 hashedVersion = keccak256(bytes(version));
        _HASHED_NAME = hashedName;
        _HASHED_VERSION = hashedVersion;
    }
```

And delete any function calls to the above two functions.

12 Authorization for cancelling an order is different for matching an order

Informational

No Change

Works as intended.

The `cancel` function simply requires `require(_msgSender() == order.maker, "not a maker");` for cancelling an order.

However orders can be authorized with `matchOrders` by using `_msgSender() == order.maker` or signatures from users or signatures from contracts.

So orders that are started with `matchOrders` using signatures can't be cancelled using signatures.

However this is fine if this is the desired functionality.

13 Gas optimize transfers in TransferExecutor

Informational

Done

Handled according to the recommendation.

The `transfer` function in the TransferExecutor contract is using proxy contracts to transfer ERC20 tokens, ERC721 tokens, ERC1155 tokens and ERC721 bundles.

Here's an estimate of the additional gas cost of using proxies to do a transfer:

1. Read the `proxies` state variable to get the address of a proxy contract - 2100 gas.
2. Make an external function call to the proxy - 2600 gas
3. The proxy contract is an upgradeable contract that works by reading a state variable (2100 gas) to get a logic contract and execute a function (2600 gas) using delegatecall -- 4700 gas
4. The proxy contract reads the `operators` state variable to check that the caller has permission to execute the transfer function - 2100 gas.

Total: 11,500 gas

The gas numbers come from EIP-2929.

13.1 Recommendation

It would be more gas efficient if the `transfer` function did the transfers for ERC20 tokens, ERC721 tokens, ERC1155 tokens directly rather than using proxy contracts.

The proxy contracts enable the system to change how the transfers work for ERC20 tokens, ERC721 tokens, ERC1155, however how transfers work for these is standardized.

The proxy contracts also enable the system to be extended to different kinds of tokens in the future. Therefore I suggest not using proxy contracts to transfer ERC20 tokens, ERC721 tokens and ERC1155 tokens since they can be implemented directly but let other future tokens use proxies.

Here is the current transfer function:

```
 function transfer(
        LibAsset.Asset memory asset,
        address from,
        address to,
        bytes4 transferDirection,
        bytes4 transferType
    ) internal override {
        if (asset.assetType.assetClass == LibAsset.ETH_ASSET_CLASS) {
            to.transferEth(asset.value);
        } else if (asset.assetType.assetClass == LibAsset.ERC20_ASSET_CLASS) {
            (address token) = abi.decode(asset.assetType.data, (address));

IERC20TransferProxy(proxies[LibAsset.ERC20_ASSET_CLASS]).erc20safeTransferFrom(IERC20Upgradeable(toke
 from, to, asset.value);
        } else if (asset.assetType.assetClass == LibAsset.ERC721_ASSET_CLASS) {
            (address token, uint tokenId) = abi.decode(asset.assetType.data, (address, uint256));
            require(asset.value == 1, "erc721 value error");

INftTransferProxy(proxies[LibAsset.ERC721_ASSET_CLASS]).erc721safeTransferFrom(IERC721Upgradeable(tok
 from, to, tokenId);
        } else if (asset.assetType.assetClass == LibAsset.ERC1155_ASSET_CLASS) {
            (address token, uint tokenId) = abi.decode(asset.assetType.data, (address, uint256));

INftTransferProxy(proxies[LibAsset.ERC1155_ASSET_CLASS]).erc1155safeTransferFrom(IERC1155Upgradeable(
 from, to, tokenId, asset.value, "");
        } else if (asset.assetType.assetClass == LibAsset.ERC721_BUNDLE_ASSET_CLASS) {
            (INftTransferProxy.ERC721BundleItem[] memory erc721BundleItems) =
abi.decode(asset.assetType.data, (INftTransferProxy.ERC721BundleItem[]));
            require(asset.value > 1 && asset.value <= maxBundleSize, "erc721 value error");

INftTransferProxy(proxies[LibAsset.ERC721_BUNDLE_ASSET_CLASS]).erc721BundleSafeTransferFrom(erc721Bun
 from, to);
        } else {
            ITransferProxy(proxies[asset.assetType.assetClass]).transfer(asset, from, to);
        }
        emit Transfer(asset, from, to, transferDirection, transferType);
    }
```

Here's the suggested change:

```
    function transfer(
        LibAsset.Asset memory asset,
        address from,
        address to,
        bytes4 transferDirection,
        bytes4 transferType
    ) internal override {
        if (asset.assetType.assetClass == LibAsset.ETH_ASSET_CLASS) {
            to.transferEth(asset.value);
        } else if (asset.assetType.assetClass == LibAsset.ERC20_ASSET_CLASS) {
            (address token) = abi.decode(asset.assetType.data, (address));
            SafeERC20.safeTransferFrom(token, from, to, asset.value);
        } else if (asset.assetType.assetClass == LibAsset.ERC721_ASSET_CLASS) {
            (address token, uint tokenId) = abi.decode(asset.assetType.data, (address, uint256));
            require(asset.value == 1, "erc721 value error");
            IERC721Upgradeable(token).safeTransferFrom( from, to, tokenId);
        } else if (asset.assetType.assetClass == LibAsset.ERC1155_ASSET_CLASS) {
            (address token, uint tokenId) = abi.decode(asset.assetType.data, (address, uint256));
            IERC1155Upgradeable(token).safeTransferFrom(from, to, tokenId, asset.value, "");
        } else if (asset.assetType.assetClass == LibAsset.ERC721_BUNDLE_ASSET_CLASS) {
            (INftTransferProxy.ERC721BundleItem[] memory erc721BundleItems) =
abi.decode(asset.assetType.data, (INftTransferProxy.ERC721BundleItem[]));
            require(asset.value > 1 && asset.value <= maxBundleSize, "erc721 value error");
            for (uint256 i = 0; i < erc721BundleItems.length; i++) {
                for (uint256 j = 0; j < erc721BundleItems[i].tokenIds.length; j++){
                    IERC721Upgradeable(erc721BundleItems[i].tokenAddress).safeTransferFrom(from,
to, erc721BundleItems[i].tokenIds[j]);
                }
            }
        } else {
            ITransferProxy(proxies[asset.assetType.assetClass]).transfer(asset, from, to);
        }
        emit Transfer(asset, from, to, transferDirection, transferType);
    }
```

Note that this change would require people to approve the UniverseMarketplace contract for transfers, rather than proxy transfer contracts.

14 Increase "runs" for Solidity optimizer

Informational

Done

Handled according to the recommendation.

The "runs" setting for the Solidity optimizer is set to 200. I suggest increasing it to get better run-time gas efficiency.

15 Unused Variable in transferRevenueSplits function

Informational

Done

Handled according to the recommendation.

In the transferRevenueSplits function the `sumBps` variable is not used for anything:

```
function transferRevenueSplits(
    LibAsset.AssetType memory matchCalculate,
    uint amount,
    address from,
    LibPart.Part[] memory revenueSplits,
    bytes4 transferDirection
) internal returns (uint) {
    uint sumBps = 0;
    uint restValue = amount;
    for (uint256 i = 0; i < revenueSplits.length && i < 5; i++) {
        uint currentAmount = amount.bp(revenueSplits[i].value);
        sumBps = sumBps.add(revenueSplits[i].value);
        if (currentAmount > 0) {
            restValue = restValue.sub(currentAmount);
            transfer(LibAsset.Asset(matchCalculate, currentAmount), from,
revenueSplits[i].account, transferDirection, REVENUE_SPLIT);
        }
    }
    return amount.sub(restValue);
}
```

15.1 Recommendation

Remove the the `sumBps` variable.

16 Disclosure

Perfect Abstractions LLC receives payment from clients (the “Clients”) for reviewing code and writing these reports (the “Reports”).

The Reports are not an accusation or endorsement of any project or team, and the Reports do not guarantee the security of any project. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. To remove any doubt, this Report is not investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the security of the project.

The Reports are created for Clients and published with their consent. The scope of our review is limited to the code or files that are specified in this report. The Solidity language remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks.