

Perfect Abstractions

Description	Niftkit V3 Audit
Copyright	Copyright © 2022 - Perfect Abstractions LLC

Table of Contents

1 Niftykit-Contracts-V3 Audit

- 1.0.1 Project description
- 1.0.2 Objectives
- 1.0.3 Scope

2 Annex

- 2.1 EIP-2535 Diamonds compliancy
- 2.2 Replay attacks possible
- 2.3 NiftyKitV3 Withdraw function could be called by anybody
- 2.4 The

I Medium Risk

3 Edition facet signature is replayable

- 3.0.1 Recommendation

II Low Risk

4 Apps can be made un-upgradable

- 4.0.1 Recommendation

5 Collection fees can be changed after mint has started

- 5.0.1 Recommendation

6 Collection parameters can be changed after mint has started

- 6.0.1 Recommendation

7 One step ownership of collections

- 7.0.1 Recommendation

8 NiftyKitV3 and NiftyKitAppRegistry initialize can be frontrun

III Informational

9 Base facet and app facets override same contracts which is prone to bugs

- 9.0.1 Recommendations

10 BaseFacet version is not used

- 10.0.1 Recommendation

11 App facets structs can be optimized

12 Some unused code can be removed

- 12.1 NiftyKitV3.sol
- 12.2 NiftyKitV3.sol
- 12.3 Unused functions

13 Low code coverage

- 13.0.1 Recommendation

14 Function parameters shadowing contract storage variables

- 14.0.1 Recommendation

15 Missing zero address checks in NiftyKitV3.sol

- 15.0.1 Recommendation

16 Lack of documentation and comments

- 16.0.1 Recommendation

17 Disclosure

1 Niftykit-Contracts-V3 Audit

[Perfect Abstractions](#) conducted a smart contract audit of Niftykit's [Niftykit-Contracts-V3](#) from 7 March 2023 to 28 March 2023.

This audit was carried out in 2 stages. The 1st stage based on the hash `1f0cd5c59429c230fd85fd97a0ea0fb0483533a4`, and the 2nd stage starting on 15 March and based on the hash `e88f77b124d513ee859ad56a106ecb88e131f7a0`, allowing further analysis of the codebase.

In this document we will mainly detail the 2nd part of the audit (git hash `e88f77b124d513ee859ad56a106ecb88e131f7a0`) because it concerns the current code version. However, we will list in the appendix certain elements highlighted and fixed in the 1st part of the audit to show the corrected vulnerabilities, certain optimizations carried out and a relevant overhaul of the design allowing compliance with the EIP-2535 Diamonds.

Auditors:

- Thibaud Catz

Audit report reviewed by Nick Mudge.

1.0.1 Project description

NiftyKit is a no-code platform for NFT creators. It allows people to create, manage and sell NFTs. This new version of NiftyKit implements EIP-2535 Diamonds, allowing it to easily add or remove functionality for NFT collections. The codebase is modular and well written.

1.0.2 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.

1.0.3 Scope

The following files were audited (hash `e88f77b124d513ee859ad56a106ecb88e131f7a0`):

- [contracts/NiftyKitAppRegistry.sol](#)
- [contracts/NiftyKitV3.sol](#)
- [contracts/apps/ape/ApeDropFacet.sol](#)
- [contracts/apps/ape/ApeDropStorage.sol](#)
- [contracts/apps/blockTokens/BlockTokensFacet.sol](#)
- [contracts/apps/drop/DropFacet.sol](#)
- [contracts/apps/drop/DropStorage.sol](#)

- [contracts/apps/edition/EditionFacet.sol](#)
- [contracts/apps/edition/EditionStorage.sol](#)
- [contracts/apps/example/ExampleFacet.sol](#)
- [contracts/apps/example/ExampleStorage.sol](#)
- [contracts/apps/operatorControls/OperatorControlsFacet.sol](#)
- [contracts/apps/royaltyControls/RoyaltyControlsFacet.sol](#)
- [contracts/diamond/BaseFacet.sol](#)
- [contracts/diamond/BaseStorage.sol](#)
- [contracts/diamond/DiamondCollection.sol](#)
- [contracts/diamond/DiamondLoupeFacet.sol](#)
- [contracts/interfaces/IDiamond.sol](#)
- [contracts/interfaces/IDiamondCut.sol](#)
- [contracts/interfaces/IDiamondLoupe.sol](#)
- [contracts/interfaces/IDropKitPass.sol](#)
- [contracts/interfaces/IERC165.sol](#)
- [contracts/interfaces/IERC173.sol](#)
- [contracts/interfaces/INiftyKitAppRegistry.sol](#)
- [contracts/interfaces/INiftyKitV3.sol](#)
- [contracts/internals/InternalERC721AUpgradeable.sol](#)
- [contracts/internals/InternalOwnable.sol](#)
- [contracts/internals/InternalOwnableRoles.sol](#)
- [contracts/internals/MinimalOwnable.sol](#)
- [contracts/internals/MinimalOwnableRoles.sol](#)
- [contracts/libraries/LibDiamond.sol](#)
- [contracts/mocks/MockERC20.sol](#)
- [contracts/mocks/MockOperator.sol](#)

2 Annex

In the 1st part of the audit from 7 March to 14 March, some issues were found and fixed.

We present the most notable ones in this annex.

2.1 EIP-2535 Diamonds compliancy

See [EIP-2535 Diamonds Implementation Points](#).

Immutable functions are external functions defined directly in a diamond proxy contract or inherited by it. The EIP-2535 Diamonds standard requires information about immutable functions be returned by the loupe functions and emitted in the DiamondCut event.

Information about immutable functions in the diamond proxy contract (`DiamondCollection.sol`) were not returned by the loupe functions and were not emitted in the DiamondCut event. This was fixed by putting all the immutable functions in a separate facet (`BaseFacet.sol`) which is now cut in the diamond proxy constructor.

The fix also made the code clearer, more modular and upgradable.

2.2 Replay attacks possible

Signatures in `NiftyKitV3.sol` and `EditionFacet.sol` were replayable on another chain. ChainId parameter has been added which fixed the issue.

2.3 NiftyKitV3 Withdraw function could be called by anybody

This was a low issue because Ether would be sent to treasury and not the transaction sender. But we can imagine a scenario where treasury would be changed (because compromised for example), and the current treasury address would frontrun the `setTreasury` function by calling `Withdraw` just before. By doing so, the compromised treasury would get the ethers in the contract before being prevented from doing it.


It has been changed to `OnlyOwner`.

2.4 The `preventTransfers` modifier could not block tokens if TransferMode is `OperatorsOnly`

It was due to some logic issue in the modifier which has been fixed.

I. Medium Risk

3 Edition facet signature is replayable

 **Medium Risk**

 **Fixed**

Fixed according to the recommendation.

In `EditionFacet.sol`, signature can be replayed in certain cases.

```
function _requireSignature(
    EditionStorage.Edition storage edition,
    uint256 editionId,
    bytes calldata signature
) internal view {
    require(
        keccak256(
            abi.encodePacked(editionId + edition.nonce, block.chainid)
        ).toEthSignedMessageHash().recover(signature) == edition.signer,
        "Invalid signature"
    );
}
```

`editionId` is incremented at each new edition. `edition.nonce` starts at zero for each new edition and can be incremented to invalidate a signature.

As the data used to make the hash is `abi.encodePacked(editionId + edition.nonce, block.chainid)`, some combinations of `editionId` and `edition.nonce` will recover to the same signer.

For example `editionId = 0` and `edition.nonce = 1`,

will be replayable if `editionId = 1` and `edition.nonce = 0`, because the `0 + 1 == 1 + 0`. So the signature could be replayed with the other combination.

A replay of the transaction on another Edition means someone could give themselves permission to mint illegitimately.

3.0.1 Recommendation

I see two options:

- Use a global nonce instead of a per Edition nonce, so that `editionId + edition.nonce` will never be repeated across the Editions
- Use `abi.encodePacked(editionId, edition.nonce, block.chainid)` instead of `abi.encodePacked(editionId + edition.nonce, block.chainid)`

II. Low Risk

4 Apps can be made un-upgradable

Low Risk

On the app facets, there is a version number which is a `uint8`:

```
struct App {
    address implementation;
    bytes4 interfaceId;
    bytes4[] selectors;
    uint8 version;
}
```

One can upgrade an app by using a superior version number:

```
require(
    version > _apps[name].version,
    "NiftyKitAppRegistry: Version must be greater than previous"
);
```

But if the `uint8` maximum value is used (255), it won't be possible to upgrade the app anymore, as new version must be greater than previous.

It could be intended behavior to provide a way to make an app un-upgradable, but in that case it's missing documentation about it.

4.0.1 Recommendation

Force version incremental values or document the fact that version numbers can be skipped and that a value of `255` will prevent an app from being upgradable.

5 Collection fees can be changed after mint has started

Low Risk

In `NiftyKitV3.sol`, a collection `feeType` and `feeRate` can be changed anytime including after minting has started.

```
...
function setRate(address collection, uint256 rate) external onlyOwner {
    Collection storage _collection = _collections[collection];
    require(_collection.exists, "Does not exist");

    _collection.feeRate = rate;
}

function setFeeType(address collection, FeeType feeType) external {
    Collection storage _collection = _collections[collection];
    require(_collection.exists, "Does not exist");
    require(IERC173(collection).owner() == _msgSender(), "Not the owner");

    _collection.feeType = feeType;
}
...
```

It means it's possible that people mint the same collection with different prices.

5.0.1 Recommendation

I would suggest not to be able to change these values, once the presale or sale has started.

6 Collection parameters can be changed after mint has started

Low Risk

In `DropFacet.sol`, `ApeDropFacet.sol` and `EditionFacet.sol`, the parameters can be changed after sale has started.

For example in `DropFacet.sol`:

```
function startSale(
    uint256 newMaxAmount,
    uint256 newMaxPerMint,
    uint256 newMaxPerWallet,
    uint256 newPrice,
    bool presale
) external onlyRolesOrOwner(BaseStorage.MANAGER_ROLE) {
    DropStorage.Layout storage layout = DropStorage.layout();
    layout._saleActive = true;
    layout._presaleActive = presale;


    layout._maxAmount = newMaxAmount;
    layout._maxPerMint = newMaxPerMint;
    layout._maxPerWallet = newMaxPerWallet;
    layout._price = newPrice;
}
```

For example we can see that `_price` can be changed after mint has started, but also `_maxPerMint`, `_maxPerWallet`.

6.0.1 Recommendation

I would suggest not to be able to change these values, once the presale or sale has started.

7 One step ownership of collections

 **Low Risk**

The functions enabling 2-step ownership have been removed from `MinimalOwnable.sol` which is inherited by the `DiamondCollection` contract.

```

/// @dev Request a two-step ownership handover to the caller.
/// The request will be automatically expire in 48 hours (172800 seconds) by default.
function requestOwnershipHandover() public payable virtual {
    unchecked {
        uint256 expires = block.timestamp + ownershipHandoverValidFor();
        /// @solidity memory-safe-assembly
        assembly {
            // Compute and set the handover slot to `expires`.
            mstore(0x0c, _HANDOVER_SLOT_SEED)
            mstore(0x00, caller())
            sstore(keccak256(0x0c, 0x20), expires)
            // Emit the {OwnershipHandoverRequested} event.
            log2(0, 0, _OWNERSHIP_HANDOVER_REQUESTED_EVENT_SIGNATURE, caller())
        }
    }
}

/// @dev Cancels the two-step ownership handover to the caller, if any.
function cancelOwnershipHandover() public payable virtual {
    /// @solidity memory-safe-assembly
    assembly {
        // Compute and set the handover slot to 0.
        mstore(0x0c, _HANDOVER_SLOT_SEED)
        mstore(0x00, caller())
        sstore(keccak256(0x0c, 0x20), 0)
        // Emit the {OwnershipHandoverCanceled} event.
        log2(0, 0, _OWNERSHIP_HANDOVER_CANCELED_EVENT_SIGNATURE, caller())
    }
}

/// @dev Allows the owner to complete the two-step ownership handover to `pendingOwner`.
/// Reverts if there is no existing ownership handover requested by `pendingOwner`.
function completeOwnershipHandover(address pendingOwner) public payable virtual onlyOwner {
    /// @solidity memory-safe-assembly
    assembly {
        // Compute and set the handover slot to 0.
        mstore(0x0c, _HANDOVER_SLOT_SEED)
        mstore(0x00, pendingOwner)
        let handoverSlot := keccak256(0x0c, 0x20)
        // If the handover does not exist, or has expired.
        if gt(timestamp(), sload(handoverSlot)) {
            mstore(0x00, _NO_HANDOVER_REQUEST_ERROR_SELECTOR)
            revert(0x1c, 0x04)
        }
        // Set the handover slot to 0.
        sstore(handoverSlot, 0)
    }
    _setOwner(pendingOwner);
}

```

This has been done, to save gas when deploying a collection but transfer of ownership is a delicate and irreversible process, it could leave a contract useless, with a two step process we add a guard against typos or bad copy/paste.

7.0.1 Recommendation

Add the functionality back or make sure it's not needed.

8 NiftyKitV3 and NiftyKitAppRegistry initialize can be frontrun

Low Risk

The functions are public which means anybody can run them (and take ownership).

```
function initialize() public initializer {
    __Ownable_init();
}
```

```
function initialize(address appRegistry_) public initializer {
    _appRegistry = appRegistry_;
    _treasury = _msgSender();
    __Ownable_init();
}
```

It is not really bad, as these functions are called during deployment, but deployer has to make sure nobody frontrun these calls during deployment. If `initialize` is frontrun, the deployer will have to redeploy because the frontrunner will get the ownership of the contracts.

III. Informational

9 Base facet and app facets override same contracts which is prone to bugs

Informational

```
contract BaseFacet is
    ERC721Upgradeable,
    MinimalOwnableRoles,
    ERC2981,
    OperatorFilterer,
    DiamondLoupeFacet
{
```

```
contract DropFacet is InternalOwnableRoles, InternalERC721Upgradeable {
```

```
contract ApeDropFacet is InternalOwnableRoles, InternalERC721Upgradeable {
```

```
contract EditionFacet is InternalOwnableRoles, InternalERC721Upgradeable {
```

They both share some inheritance (because `InternalOwnableRoles` is an internal modified version of `MinimalOwnableRoles` and `InternalERC721Upgradeable` is an internal modified version of `ERC721Upgradeable`).

This has been done so that app facets can easily call functions and use storage from `ERC721A` and `OwnableRoles`.

But this design requires attention to the risks of having an overridden function in `BaseFacet` which is not overridden in an app facet. In that case, when a function is called from an app facet it would execute original code instead of overridden one which is undesired and leads to bugs.

A bug has been found during the audit which was caused exactly by this fact.

The function:

```
function _startTokenId() internal pure override returns (uint256) {
    return 1;
}
```

Is an override in `BaseFacet`, allowing to have token ids starting at 1 instead of 0.

But the function was not overridden in facets, and the consequence was that max supply could not be reached.

Solution to the bug was to implement the same override in each app facet.

Fortunately, bug was found and fixed.

9.0.1 Recommendations

There are a few different ways to share code between facets. We recommend these ways:

- Write internal functions in Solidity libraries and import and call those functions in facets.
- Put common internal functions in a contract that is inherited by multiple facets. Internal functions defined with the `virtual` keyword can be overridden. Consider not using the `virtual` keyword to ensure shared internal functions are the same between facets.

More information about sharing code between facets and ways to do it are in this article: [How to Share Functions Between Facets of a Diamond](#)

10 BaseFacet version is not used

Informational

The `BaseFacet` has a version number and can be upgraded.

```
function setBase(  
    address implementation,  
    bytes4[] calldata interfaceIds,  
    bytes4[] calldata selectors,  
    uint8 version  
) external onlyOwner {  
    _base = Base({  
        implementation: implementation,  
        interfaceIds: interfaceIds,  
        selectors: selectors,  
        version: version  
    });  
}
```

But the version is only used to be stored in the `DiamondCollection` layout:

```
layout._baseVersion = base.version;
```

So any version can be set, a new `BaseFacet` could have same version as previous or a lower number, for example.

10.0.1 Recommendation

Make sure that an upgrade of `BaseFacet` has a superior version (as done for app facets).

11 App facets structs can be optimized

In [ERC721A.sol source code](#), we can see the following assumption:

```
- An owner cannot have more than  $2^{64} - 1$  (max value of uint64) of supply.
```

So as the max number of tokens per wallet is 2^{64} ,

in `DropStorage.sol`:

```
struct Layout {
    mapping(address => uint256) _mintCount;
    bytes32 _merkleRoot;
    uint256 _dropRevenue;
    // Sales Parameters
    uint256 _maxAmount;
    uint256 _maxPerMint;
    uint256 _maxPerWallet;
    uint256 _price;
    // States
    bool _presaleActive;
    bool _saleActive;
}
```

and in `EditionStorage.sol`:

```
struct Edition {
    string tokenURI;
    bytes32 merkleRoot;
    uint256 price;
    uint256 quantity;
    uint256 maxQuantity;
    uint256 maxPerWallet;
    uint256 maxPerMint;
    uint256 nonce;
    address signer;
    bool active;
}
```

`_maxPerMint` and `_maxPerWallet`, could be `uint64`.

It would save one slot in each struct which would save gas when accessing the structs.

If the change is made, the functions getting/setting the values should be modified too, for example in `EditionFacet.sol`:

```
function createEdition(
    string memory tokenURI,
    uint256 price,
    uint256 maxQuantity,
    uint256 maxPerWallet,
    uint256 maxPerMint
```

12 Some unused code can be removed

Informational

12.1 NiftyKitV3.sol

`import {ClonesUpgradeable};` can be removed as it's not used.
`import {IDropKitPass};` can be removed as it's not used.

12.2 NiftyKitV3.sol

Here: `contract NiftyKitV3 is INiftyKitV3, Initializable, OwnableUpgradeable`

`Initializable` can be removed as it's inherited by `OwnableUpgradeable`

12.3 Unused functions

Here's a list of unused code. Removing this code will:

- Help code readability
- Increase control about what the code does or not

✘ `InternalERC721AUpgradeable._nextTokenId()` (`contracts/internals/InternalERC721AUpgradeable.sol`) is never used

✘ `InternalERC721AUpgradeable._baseURI()` (`contracts/internals/InternalERC721AUpgradeable.sol`) is never used

✘ `MinimalOwnableRoles._checkOwnerOrRoles(uint256)` (`contracts/internals/MinimalOwnableRoles.sol`) is never used

✘ `InternalERC721AUpgradeable._unpackedOwnership(uint256)` (`contracts/internals/InternalERC721AUpgradeable.sol`) is never used

✘ `InternalOwnable._setOwner(address)` (`contracts/internals/InternalOwnable.sol`) is never used

✘ `InternalERC721AUpgradeable._approve(address,uint256)` (`contracts/internals/InternalERC721AUpgradeable.sol`) is never used

✘ `InternalOwnableRoles._grantRoles(address,uint256)` (`contracts/internals/InternalOwnableRoles.sol`) is never used

✘ `InternalERC721AUpgradeable._exists(uint256)` (`contracts/internals/InternalERC721AUpgradeable.sol:438-445`) is never used

- ✘ InternalERC721Upgradeable._ownershipAt(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._toString(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ MinimalOwnableRoles._checkRoles(uint256) (contracts/internals/MinimalOwnableRoles.sol) is never used
- ✘ InternalERC721Upgradeable._setExtraDataAt(uint256,uint24) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._totalBurned() (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable.__ERC721A_init_unchained(string,string) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ LibDiamond.enforcelsContractOwner() (contracts/libraries/LibDiamond.sol) is never used
- ✘ InternalERC721Upgradeable._numberBurned(address) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._burn(uint256,bool) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._isApprovedForAll(address,address) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._getApprovedSlotAndAddress(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._approve(address,uint256,bool) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalOwnableRoles._checkOwnerOrRoles(uint256) (contracts/internals/InternalOwnableRoles.sol) is never used
- ✘ InternalERC721Upgradeable._totalMinted() (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalOwnableRoles._checkRoles(uint256) (contracts/internals/InternalOwnableRoles.sol) is never used
- ✘ InternalERC721Upgradeable._ownershipOf(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._isSenderApprovedOrOwner(address,address,address) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._burn(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used

- ✘ InternalERC721Upgradeable._mintERC2309(address,uint256)
(contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._packedOwnershipOf(uint256)
(contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable._ownerOf(uint256) (contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ LibDiamond.contractOwner() (contracts/libraries/LibDiamond.sol) is never used
- ✘ InternalOwnableRoles._removeRoles(address,uint256) (contracts/internals/InternalOwnableRoles.sol) is never used
- ✘ InternalERC721Upgradeable._initializeOwnershipAt(uint256)
(contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalOwnable._initializeOwner(address) (contracts/internals/InternalOwnable.sol) is never used
- ✘ InternalERC721Upgradeable._numberMinted(address)
(contracts/internals/InternalERC721Upgradeable.sol) is never used
- ✘ InternalERC721Upgradeable.__ERC721A_init(string,string)
(contracts/internals/InternalERC721Upgradeable.sol) is never used

13 Low code coverage

Informational

Code coverage is low on certain files (mainly `internal` folder).

- InternalERC721AUpgradeable.sol 20.54%
- InternalOwnable.sol 0%
- InternalOwnableRoles.sol 50%
- MinimalOwnable.sol 0%
- MinimalOwnableRoles.sol 22.22%
- RoyaltyControlsFacet.sol 0%

Some of these are critical because their goal is:

- To assure correct ownership of the contracts
- The ERC721A part of the app facets

13.0.1 Recommendation

Add some tests to cover the whole code base.

14 Function parameters shadowing contract storage variables

Informational

For example in `BaseFacet.sol`,

```
function installApp(bytes32 name) external onlyOwner {
    _installApp(name, address(0), "");
}

function installApp(bytes32 name, bytes memory data) external onlyOwner {
    _installApp(name, address(this), data);
}

function removeApp(bytes32 name) external onlyOwner {
    _removeApp(name, address(0), "");
}

function removeApp(bytes32 name, bytes memory data) external onlyOwner {
    _removeApp(name, address(this), data);
}
```

`name` parameter shadows the `ERC721Upgradable.name` storage variable. It works without issues and it's not a problem by itself, but it could mislead code readers or developers.

14.0.1 Recommendation

Consider using different function variable names that don't shadow storage variables.

15 Missing zero address checks in NiftyKitV3.sol

Informational

```
function initialize(address appRegistry_) public initializer {
    _appRegistry = appRegistry_;
    _treasury = _msgSender();
    __Ownable_init();
}

function setTreasury(address newTreasury) external onlyOwner {
    _treasury = newTreasury;
}

function setSigner(address signer) external onlyOwner {
    _signer = signer;
}
```

These function don't check that the input address is not zero. It's better not to be able to set a 0 address, than to discover that it has been set by error.

15.0.1 Recommendation

Add zero-address check to these functions.

16 Lack of documentation and comments

Informational

Lack of function-level documentation (natspec) and an absence of comments in the code base.

16.0.1 Recommendation

Introducing comprehensive function-level documentation and comments throughout the code base will make it significantly easier to understand, reason about, maintain and update.

17 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.