CONSENSYS Diligence
AUDITS FUZZING SCRIBBLE ABOUT

Stakewise - v3 Vaults / EthFoxVault

Date	March 2024
Auditors	Martin Ortner, Valentin Quelquejay

1 Executive Summary

This report presents the results of our engagement with **StakeWise** and **Consensys** to review **v3 Vaults/EthFoxVault** implementation for **Consensys**.

The review was conducted over three weeks, from **January 29, 2024**, to **March 1, 2024**. A total of 30 person-days were spent.

Stakewise V3 is a liquid staking protocol set to be utilized by Consensys as the underlying system for Metamask Staking. This modular system comprises a collection of customizable modules that define the behavior of a staking vault. While modularity is a valuable concept, it’s crucial to balance it with simplicity, especially when it comes to managing complexity. In some cases, overly modular designs can inadvertently lead to complex interdependencies among modules, resembling a tangled inheritance tree rather than a clear, manageable structure.

The system relies on several privileged accounts that can alter its behavior. Furthermore, the protocol depends on off-chain components and oracles for multiple tasks, including validator authorization/exit and reward updates.

A registry owner can register Vaults out-of-band bypassing assurances provided by VaultFactories. The keeper admin can add/remove oracles at their discretion. Both are refered to as the network governor in the task scripts. While the goal of the oracles is to minimize the trust asumptions, trusting a majority of the oracles - and therefore also the off-chain components - is critical.

Configurative parameters are mostly unsanitized at the smart-contract level. Therefore, it is critical to ensure that the smart contracts are intialized with safe parameters before utilizing them. Initialization is mostly unprotected allowing anyone to front-run initialization as, specifically `EthFoxVault`, is not deployed by a factory and existing scripts suggest that deployment and initialization not be performed in the same transaction.

To ensure a high level of trust and security, it is essential for the admins to be a DAO, as the admins hold critical powers that can directly impact users' funds. Furthermore, any call of admin functionality must be thoroughly validated and verified by the DAO members to avoid potentially malicious system changes to be performed (Governor approved contract upgrades, vault additions, keepers).

2 Scope

Our review focused on the commit hash eb908436fd604936da38a111389408c94a7218c8 with the main focus on `EthFoxVault.sol` and surrounding functionality. The list of files in scope can be found in the Appendix.

2.1 Objectives

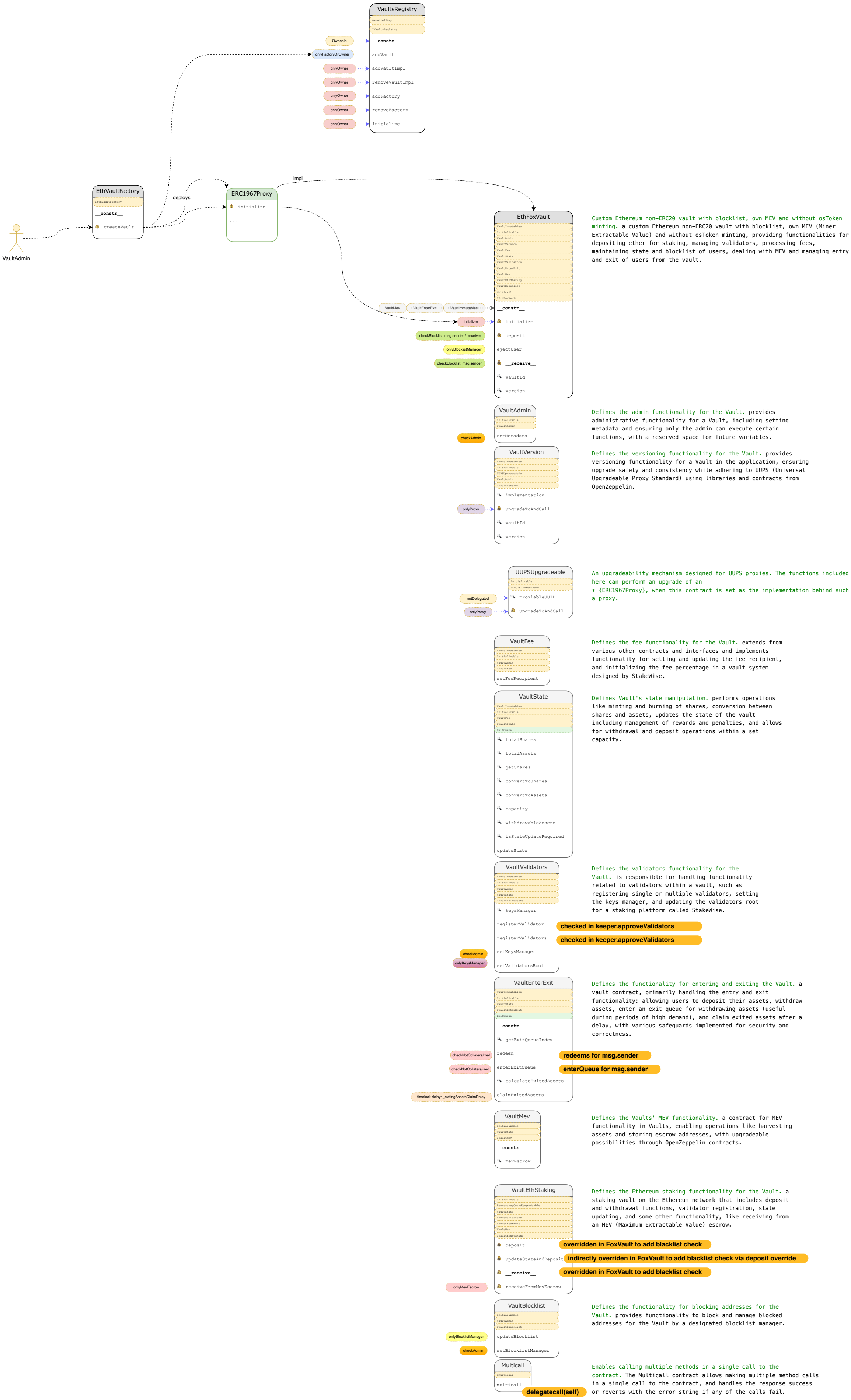
Together with the **StakeWise** and **Consensys** teams, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

3.1 Architecture Overview

Below we provide a high-level diagram depicting the main components of the system and their dependencies.



Custom Ethereum non-ERC20 vault with blocklist, own MEV and without osToken minting. a custom Ethereum non-ERC20 vault with blocklist, own MEV (Miner Extractable Value) and without osToken minting, providing functionalities for depositing ether for staking, managing validators, processing fees, maintaining state and blocklist of users, dealing with MEV and managing entry and exit of users from the vault.

Defines the admin functionality for the Vault. provides administrative functionality for a Vault, including setting metadata and ensuring only the admin can execute certain functions, with a reserved space for future variables.

Defines the versioning functionality for the Vault. provides versioning functionality for a Vault in the application, ensuring upgrade safety and consistency while adhering to UUPS (Universal Upgradeable Proxy Standard) using libraries and contracts from OpenZeppelin.

An upgradeability mechanism designed for UUPS proxies. The functions included here can perform an upgrade of an
* {ERC1967Proxy}, when this contract is set as the implementation behind such a proxy.

Defines the fee functionality for the Vault. extends from various other contracts and interfaces and implements functionality for setting and updating the fee recipient, and initializing the fee percentage in a vault system designed by StakeWise.

Defines Vault's state manipulation. performs operations like minting and burning of shares, conversion between shares and assets, updates the state of the vault including management of rewards and penalties, and allows for withdrawal and deposit operations within a set capacity.

Defines the validators functionality for the Vault. is responsible for handling functionality related to validators within a vault, such as registering single or multiple validators, setting the keys manager, and updating the validators root for a staking platform called StakeWise.

Defines the functionality for entering and exiting the Vault. a vault contract, primarily handling the entry and exit functionality: allowing users to deposit their assets, withdraw assets, enter an exit queue for withdrawing assets (useful during periods of high demand), and claim exited assets after a delay, with various safeguards implemented for security and correctness.

Defines the Vaults' MEV functionality. a contract for MEV functionality in Vaults, enabling operations like harvesting assets and storing escrow addresses, with upgradeable possibilities through OpenZeppelin contracts.

Defines the Ethereum staking functionality for the Vault. a staking vault on the Ethereum network that includes deposit and withdrawal functions, validator registration, state updating, and some other functionality, like receiving from an MEV (Maximum Extractable Value) escrow.

Defines the functionality for blocking addresses for the Vault. provides functionality to block and manage blocked addresses for the Vault by a designated blocklist manager.

Enables calling multiple methods in a single call to the contract. The Multicall contract allows making multiple method calls in a single call to the contract, and handles the response success or reverts with the error string if any of the calls fail.

3.2 Key Security Specifications

- Security parameters are not sanitized at the contract level. Therefore, it is critical to ensure that the contracts' parameters are initialized with safe defaults. For instance, improperly configured exit time constraints might open the door to unfair arbitrage opportunities; a consensus threshold for oracles configured too low might decrease the security of the system.

- The project relies on oracles for multiple operations including authorizing/exiting validators, or posting rewards. It is critical that a majority of oracles behave properly.

- The project relies on privileged account(s) to carry out several critical actions such as controlling the blocklist, setting the feeRecipient, authorizing upgrades, or publishing the validator tree merkle root. It is critical that these accounts are secured, ideally as multisig accounts, since the compromise of any one of these accounts could have severe consequences on the protocol.

- Consensys `EthFoxVault` specificities:
  - Vault shares are non-transferrable.
  - The vault has a blacklisting/ejection mechanism.
  - The vault has its own MEV escrow contract.
  - Both oracles and Consensys can trigger the exit of validators.
  - The vault is upgradeable but the upgradability of the vault smart contract is constrained by a dual governance model where both the vault owner (Consensys) and the Stakewise DAO have to sign an on-chain transaction.
  - Fees can only be changed by upgrading the contracts.
  - The vault owner can add/remove oracles meaning a malicious owner could carry out malicious actions such as messing with rewards, or preventing funds from being redeemed.
  - Consensys is responsible for provisioning new validators when enough funds are present in the contract. However, Consensys has to get oracles' approval to register new validators.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 VaultsRegistry Allows Owner to Bypass VaultFactory

Medium Acknowledged

Resolution

Acknowledged - By Design. While a registry owner (StakeWise DAO) can register factories, and these factories, in turn, can register vaults, adding vaults out-of-band doesn't increase trust in the system.

Description

Creating a Vault with an approved VaultFactory ensures that only approved implementations of Vaults can be registered with the system. For example, when a Vault Owner creates a new Vault via `VaultFactory.createVault()` the function deploys a proxy pointing to a fixed implementation. Additionally, the newly added Vault address is registered with the VaultRegistry. Only approved factories are allowed to register Vaults with the `VaultRegistry.addVault()`.

However, the guarantee that only approved Factories (with approved implementations) can register Vaults is undermined by permissions the VaultRegistry owner has. They can unilaterally register Vaults that have not been created by the VaultFactory.

It is not clear if the registered Vault is actually a vault contract (or even EOA), nor if it's been properly initialized within one transaction (see issue 4.5) as it can be registered out of band by the owner.

Examples

contracts/vaults/VaultsRegistry.sol:L31-L37

```
/// @inheritdoc IVaultsRegistry
function addVault(address vault) external override {
  if (!factories[msg.sender] && msg.sender != owner()) revert Errors.Acces

  vaults[vault] = true;
  emit VaultAdded(msg.sender, vault);
}
```

Recommendation

Ensure and provide guarantees on the origin of Vaults by enforcing that they've been created with an approved factory. If the owner is a multi-sig or DAO, ensure that everyone understands the implications of allowing the owner to add vaults to the registry out-of-band (`SharedMevRewards`) and the scrutiny required to avoid that a malicious Vault is added to the registry.

4.2 `KeeperRewards.canUpdateRewards()` and `VaultMev._harvestAssets()` Can Theoretically Overflow Medium Acknowledged

Resolution

Acknowledged - By Design. The Keeper contract is immutable and has already been deployed/verified here with rewardsDelay set to 43200

Description

- `KeeperRewards.canUpdateRewards()`

The inline comment mentions that the unchecked block cannot overflow as “lastRewardsTimestamp & rewardsDelay are uint64”. Yet, `rewardsDelay` is a uint256 meaning the result of `lastRewardsTimestamp + rewardsDelay` could overflow if `rewardsDelay > 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF - lastRewardsTimestamp`. If this operation resulted in an overflow, `canUpdateRewards()` would likely return `true` in invalid scenarios. This should not happen if the contract is initialized with reasonable values. However, in case the contract is incorrectly initialized or upgraded, this operation could overflow and lead to unintended effects.

contracts/keeper/KeeperRewards.sol:L132-L137

```
function canUpdateRewards() public view override returns (bool) {
  unchecked {
    // cannot overflow as lastRewardsTimestamp & rewardsDelay are uint64
    return lastRewardsTimestamp + rewardsDelay < block.timestamp;
  }
}
```

contracts/keeper/KeeperRewards.sol:L33

```
uint256 public immutable override rewardsDelay;
```

- `VaultMev._harvestAssets()`

An integer overflow may occur with an unsafe cast to int256(uint256) for values exceeding uint256.MAX/2+1. Although the likelihood of this happening within the current system and chain (ETH/Mainnet) is extremely low, requiring the contract to return an asset equivalent to over half of the ETH total supply, which is practically unfeasible. However, if this contract is reused, either wholly or partially, on different chains with different configurations or custom tokens, there could be a heightened risk. Therefore, to ensure robust security practices, it is advisable to invest a small amount of gas by implementing `SafeCast` to enforce secure coding standards.

contracts/vaults/modules/VaultMev.sol:L59-L63

```

  // execution rewards are always equal to what was accumulated in own MEV e
  return (totalAssetsDelta + int256(IOwnMevEscrow(_mevEscrow).harvest()),
}
```

contracts/vaults/ethereum/mev/OwnMevEscrow.sol:L23-L32

```
function harvest() external returns (uint256 assets) {
  if (msg.sender != vault) revert Errors.HarvestFailed();

  assets = address(this).balance;
  if (assets == 0) return 0;

  emit Harvested(assets);
  // slither-disable-next-line arbitrary-send-eth
  IVaultEthStaking(msg.sender).receiveFromMevEscrow{value: assets}();
}
```

Recommendation

For `KeeperRewards.canUpdateRewards()`, use a `uint64` for `rewardsDelay`, remove the unchecked block or ensure `rewardsDelay` is within reasonable bounds to prevent any overflow. Ensure that comments are accurate. For `VaultMev._harvestAssets()` and other unsafe casts, use the `SafeCast` library to revert on over/underflows.

4.3 Inconsistent Interface Between Similar Functionality: Whitelist / Blocklist Medium Acknowledged

Resolution

The client acknowledged the issue and provided the following statement: Acknowledged - Feature Enhancement

Description

`VaultBlocklist` and `VaultWhitelist` implement similar functionality. However, their handling from a management perspective is quite different.

- `setBlocklistManager` and `setWhitelister` implementations are the same. However, we would suggest using the same terminology for both functions (i.e. “BlocklistManager”, “WhitelistManager”).
- `__VaultBlocklist_init` takes a `blocklistManager` and stores it, while `__VaultWhitelist_init` also whitelists the `whitelistManager`. Given that a

whitelister should only be used as a management account it is unclear why it needs to be whitelisted for e.g. token transfers?

- Blocklist exports a `_checkBlocklist()` method, while Whitelist requires callers to check the `whitelistedAccounts` mapping directly.
- `updateBlocklist()` silently proceeds even if the same address is blocklisted twice while `_updateWhitelist()` would revert

The differing management approaches between `VaultBlocklist` and `VaultWhitelist` introduce inconsistency and potential confusion for developers and users. This lack of uniformity in functionality and handling could lead to errors in implementation and management of blocklisted and whitelisted accounts, impacting the overall security and usability of the system.

Examples

- Blocklist `updateBlocklist()`:

contracts/vaults/modules/VaultBlocklist.sol:L23-L29

```
function updateBlocklist(address account, bool isBlocked) public virtual o
  if (msg.sender != blocklistManager) revert Errors.AccessDenied();
  if (blockedAccounts[account] == isBlocked) return;

  blockedAccounts[account] = isBlocked;
  emit BlocklistUpdated(msg.sender, account, isBlocked);
}
```

- Whitelist `updateWhitelist()`:

contracts/vaults/modules/VaultWhitelist.sol:L23-L27

```
function updateWhitelist(address account, bool approved) external override
  if (msg.sender != whitelister) revert Errors.AccessDenied();
  _updateWhitelist(account, approved);
}
```

contracts/vaults/modules/VaultWhitelist.sol:L34-L43

```
/**
 * @notice Internal function for updating whitelist
 * @param account The address of the account to update
 * @param approved Defines whether account is added to the whitelist or remo
 */
function _updateWhitelist(address account, bool approved) private {
  if (whitelistedAccounts[account] == approved) revert Errors.WhitelistAlr
  whitelistedAccounts[account] = approved;
  emit WhitelistUpdated(msg.sender, account, approved);
}
```

Recommendation

Consider using the same or a similar interface for both Blocklist and Whitelist features.

4.4 User Can Prevent Blocklist Manager From Ejecting Them by Reverting on ETH Transfer Medium Acknowledged

Resolution

Acknowledged - Can be avoided by collateralising the vault

Description

In the scenario where the Blocklist Manager intends to remove a user from the vault, particularly when the vault is not collateralized, the Blocklist Manager initiates the removal process by invoking the `ejectUser()` function within the `EthFoxVault` contract. Subsequently, this action triggers the execution of the `redeem()` path for the respective user.

contracts/vaults/ethereum/custom/EthFoxVault.sol:L87-L103

```
/// @inheritdoc IEthFoxVault
function ejectUser(address user) external override {
  // add user to blocklist
  updateBlocklist(user, true);

  // fetch shares of the user
  uint256 userShares = _balances[user];
  if (userShares == 0) return;

  if (_isCollateralized()) {
    // send user shares to exit queue
    _enterExitQueue(user, userShares, user);
  } else {
    // redeem user shares
    _redeem(user, userShares, user);
  }
}
```

contracts/vaults/modules/VaultEnterExit.sol:L164-L190

```
function _redeem(
  address user,
  uint256 shares,
  address receiver
) internal returns (uint256 assets) {
  _checkNotCollateralized();
  if (shares == 0) revert Errors.InvalidShares();
  if (receiver == address(0)) revert Errors.ZeroAddress();

  // calculate amount of assets to burn
  assets = convertToAssets(shares);
  if (assets == 0) revert Errors.InvalidAssets();

  // reverts in case there are not enough withdrawable assets
  if (assets > withdrawableAssets()) revert Errors.InsufficientAssets();

  // update total assets
  _totalAssets -= SafeCast.toUint128(assets);

  // burn owner shares
  _burnShares(user, shares);

  // transfer assets to the receiver
  _transferVaultAssets(receiver, assets);

  emit Redeemed(user, receiver, assets, shares);
}
```

This operation eventually leads to the execution of `_transferVaultAssets()`, a function responsible for facilitating a low-level value-contract-call to the designated `recipient`. In the event of an unsuccessful call, the operation reverts, ensuring the integrity of the transaction.

contracts/vaults/modules/VaultEthStaking.sol:L124-L130

```
/// @inheritdoc VaultEnterExit
function _transferVaultAssets(
  address receiver,
  uint256 assets
) internal virtual override nonReentrant {
  return Address.sendValue(payable(receiver), assets);
}
```

```
function sendValue(address payable recipient, uint256 amount) internal
    if (address(this).balance < amount) {
        revert AddressInsufficientBalance(address(this));
    }

    (bool success, ) = recipient.call{value: amount}("");
    if (!success) {
        revert FailedInnerCall();
    }
}
```

Upon sending value to the designated `recipient`, the recipient's fallback function, if available, is invoked, effectively transferring control to the recipient. At this juncture, the recipient possesses the option to revert the call within their fallback function. Should the recipient choose to revert the call, the outer call initiated at `sendValue()` would also revert accordingly.

In scenarios where the vault is not collateralized, the `recipient` can evade being ejected from the vault by intentionally reverting within their fallback function. This strategic maneuver allows the `recipient` to thwart the expulsion attempt initiated by the `EthFoxVault` contract during the redemption process.

Recommendation

Change from push to pull transfers. Document that a user can still be ejected by collateralizing the vault.

4.5 Front-Running Vulnerability During Initialization of Vault Contracts Medium ✓ Fixed

Resolution

- The client mentioned the issue was fixed in this PR

Description

Vault contracts like `EthFoxVault` are deployed as proxy contracts, which point to a specific implementation. The contract follows the OpenZeppelin (OZ) `Initializable` pattern without permissioning the `initialize()` function. This pattern necessitates that the deployment of the proxy contract and its initialization occur immediately, within the same transaction. Failure to do so exposes the contract to the risk of front-running, where any user could potentially manipulate the initialization process and claim administrative control over the contract.

contracts/vaults/ethereum/custom/EthFoxVault.sol:L49-L75

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor(
  address _keeper,
  address _vaultsRegistry,
  address _validatorsRegistry,
  address sharedMevEscrow,
  uint256 exitedAssetsClaimDelay
)
  VaultImmutables(_keeper, _vaultsRegistry, _validatorsRegistry)
  VaultEnterExit(exitedAssetsClaimDelay)
  VaultMev(sharedMevEscrow)
{
  _disableInitializers();
}

/// @inheritdoc IEthFoxVault
function initialize(bytes calldata params) external payable virtual overri
  EthFoxVaultInitParams memory initParams = abi.decode(params, (EthFoxVaul
  __EthFoxVault_init(initParams);
  emit EthFoxVaultCreated(
    initParams.admin,
    initParams.ownMevEscrow,
    initParams.capacity,
    initParams.feePercent,
    initParams.metadataIpfsHash
  );
}
```

Users deploying contracts via the `VaultFactory` are inherently protected, as both the deployment of the proxy and the `initialization` are executed within the same transaction.

contracts/vaults/ethereum/EthVaultFactory.sol:L38-L58

```
/// @inheritdoc IEthVaultFactory
function createVault(
  bytes calldata params,
  bool isOwnMevEscrow
) external payable override returns (address vault) {
  // create vault
  vault = address(new ERC1967Proxy(implementation, ''));

  // create MEV escrow contract if needed
  address _mevEscrow;
  if (isOwnMevEscrow) {
    _mevEscrow = address(new OwnMevEscrow(vault));
    // set MEV escrow contract so that it can be initialized in the Vault
    ownMevEscrow = _mevEscrow;
  }

  // set admin so that it can be initialized in the Vault
  vaultAdmin = msg.sender;

  // initialize Vault
  IEthVault(vault).initialize{value: msg.value}(params);
```

However, it's worth noting that the task scripts located in the `./tasks/` directory do not utilize the `VaultFactory`, `MultiCall`, or Hardhat code to deploy and `initialize` in the same transaction. Consequently, vaults deployed via these task scripts are vulnerable to front-running by any party.

Examples

- FoxVault - vulnerable

tasks/eth-full-deploy.ts:L255-L278

```
const foxVault = foxVaultFactory.attach(foxVaultAddress)

// Initialize EthFoxVault
const ownMevEscrowFactory = await ethers.getContractFactory('OwnMevEscrow'
const ownMevEscrow = await ownMevEscrowFactory.deploy(foxVaultAddress)
await callContract(
  foxVault.initialize(
    ethers.AbiCoder.defaultAbiCoder().encode(
      [
        'tuple(address admin, address ownMevEscrow, uint256 capacity, uint
      ],
      [
        [
          networkConfig.foxVault.admin,
          await ownMevEscrow.getAddress(),
          networkConfig.foxVault.capacity,
          networkConfig.foxVault.feePercent,
          networkConfig.foxVault.metadataIpfsHash,
        ],
      ]
    ),
    { value: networkConfig.securityDeposit }
  )
)
```

- Keeper, VaultsRegistry - not vulnerable due to `function initialize() onlyOwner`

tasks/eth-full-deploy.ts:L319-L324

```
// transfer ownership to governor
await callContract(vaultsRegistry.initialize(networkConfig.governor))
console.log('VaultsRegistry ownership transferred to', networkConfig.gover

await callContract(keeper.initialize(networkConfig.governor))
console.log('Keeper ownership transferred to', networkConfig.governor)
```

- Test Suite does not even use `Factory.createVault` but deploy-initializes manually

test/shared/fixtures.ts:L715-L731

```
await vault.initialize(
  ethers.AbiCoder.defaultAbiCoder().encode(
    [
      'tuple(address admin, address ownMevEscrow, uint256 capacity, uint16
    ],
    [
      [
        adminAddr,
        await ownMevEscrow.getAddress(),
        vaultParams.capacity,
        vaultParams.feePercent,
        vaultParams.metadataIpfsHash,
      ],
    ]
  ),
  { value: SECURITY_DEPOSIT }
)
```

Recommendation

Note: According to the StakeWise team there is no intention to create a factory for the `EthFoxVault`. We recommend implementing safe deploy&initialize procedures instead of solely relying on verifying contract parameterization after deployment.

- Change the task scripts to use hardhat deploy & initialize code that performs all action in a single transaction.
- Switch to multicall deployment & initialize pattern.
- Enforce permission on `initialize()` to `onlyOwner` as seen with `Keeper` and `VaultsRegistry`
- Monitor and verify correct contract parameterisation after deployment

4.6 Consider Emitting a Specific Event in `ejectUser()`

Minor ✓ Fixed

Resolution

Fixed in this PR

Description

The function `ejectUser()` in `EthFoxVault` allows the BlocklistManager to ban and eject a user from the system. Consider making this function emit a specific event for transparency and auditability reasons.

Examples

contracts/vaults/ethereum/custom/EthFoxVault.sol:L87-L88

```
/// @inheritdoc IEthFoxVault
function ejectUser(address user) external override {
```

Recommendation

Emit a specific event when a user is ejected from the system.

4.7 VaultFactory/VaultsRegistry - Should Check That New Implementation Is a Contract Minor Acknowledged

Resolution

The client provided the following statement: Acknowledged - The necessary checks are already performed during the proxy deployment/upgrade process here

Description

Currently, the `addVaultImpl()` function in the `VaultsRegistry` contract lacks validation that address of `newImpl` actually has code. If an invalid address, devoid of code, is set as an implementation, vaults wishing to upgrade to that implementation will not be able to upgrade.

Example

- `EthVaultFactory.implementation` may not be a contract. This will cause a revert in `createVault` when `ERC1967Proxy()` is instantiated.

contracts/vaults/ethereum/EthVaultFactory.sol:L28-L36

```
/**
 * @dev Constructor
 * @param _implementation The implementation address of Vault
 * @param vaultsRegistry The address of the VaultsRegistry contract
 */
constructor(address _implementation, IVaultsRegistry vaultsRegistry) {
  implementation = _implementation;
  _vaultsRegistry = vaultsRegistry;
}
```

- `VaultsRegistry.addVaultImpl` does not validate `newImpl`. This will cause an implicit revert in `UUPSUpgradeable._upgradeToAndCallUUPS` during upgrade.

contracts/vaults/VaultsRegistry.sol:L40-L44

```
function addVaultImpl(address newImpl) external override onlyOwner {
  if (vaultImpls[newImpl]) revert Errors.AlreadyAdded();
  vaultImpls[newImpl] = true;
  emit VaultImplAdded(newImpl);
}
```

Recommendation

Enforce a check on `newImpl.code.size > 0` early on.

4.8 Consider Using `abi.encodeCall` Instead of Low-Level `bytes4(keccak(..))` and `abi.encodeWithSelector`

Minor Acknowledged

Resolution

Acknowledged - By Design

Description

Consider using Solidity contract type interfaces for building low-level contract calls with arguments, instead of constructing them manually from function declaration strings.

Examples

contracts/vaults/modules/VaultVersion.sol:L26-L27

```
bytes4 private constant _initSelector = bytes4(keccak256('initialize(bytes
```

contracts/vaults/modules/VaultVersion.sol:L34-L39

```
function upgradeToAndCall(
  address newImplementation,
  bytes memory data
) public payable override onlyProxy {
  super.upgradeToAndCall(newImplementation, abi.encodeWithSelector(_initSe
}
```

Recommendation

Use `abi.encodeCall(IVault.initialize, (bytes data))` instead of falling back to low-level function selector calculations.

4.9 Consider Reverting on Ineffective Calls Minor Acknowledged

Resolution

Acknowledged - By Design. Reducing the byte code size of the vault contract means that these validations can be omitted.

Description

The function `_setWhitelister()` (resp. `_setBlocklistManager()`) does not revert if the admin attempts to set the same `_whitelister` (resp. `_blocklistManager`). This behavior might hide mistakes. Additionally, the functions emits a `WhitelisterUpdated` (resp. `BlocklistManagerUpdated`) event even though the `_whitelister` (resp. `_blocklistManager`) hasn't been modified.

Examples

- Blocklist

contracts/vaults/modules/VaultBlocklist.sol:L31-L35

```
/// @inheritdoc IVaultBlocklist
function setBlocklistManager(address _blocklistManager) external override
  _checkAdmin();
  _setBlocklistManager(_blocklistManager);
}
```

contracts/vaults/modules/VaultBlocklist.sol:L49-L53

```
function _setBlocklistManager(address _blocklistManager) private {
  // update blocklist manager address
  blocklistManager = _blocklistManager;
  emit BlocklistManagerUpdated(msg.sender, _blocklistManager);
}
```

- Whitelist

contracts/vaults/modules/VaultWhitelist.sol:L28-L32

```
/// @inheritdoc IVaultWhitelist
function setWhitelister(address _whitelister) external override {
  _checkAdmin();
  _setWhitelister(_whitelister);
}
```

contracts/vaults/modules/VaultWhitelist.sol:L45-L53

```
/**
 * @dev Internal function for updating the whitelister externally or from th
 * @param _whitelister The address of the new whitelister
 */
function _setWhitelister(address _whitelister) private {
  // update whitelister address
  whitelister = _whitelister;
  emit WhitelisterUpdated(msg.sender, _whitelister);
}
```

Recommendation

Consider reverting the `_setWhitelister()` (resp. `_setBlocklistManager()`) function execution in case one attempts to set the same `_whitelister` (resp. `_blocklistManager`).

4.10 Vault Admin Is Non-Transferable Minor Acknowledged

Resolution
Acknowledged - By Design

Description

The Vault admin is set on initialization.

- There is no `address(0)` check preventing no admin from being set at initialization.
- Admin access can only be set on initialization and not be transferred (2-step). This may leave the Vault vulnerable and the admins unable to react in case of a vault admin address compromise.

contracts/vaults/modules/VaultAdmin.sol:L24-L34

```
/**
 * @dev Initializes the VaultAdmin contract
 * @param _admin The address of the Vault admin
 */
function __VaultAdmin_init(
  address _admin,
  string memory metadataIpfsHash
) internal onlyInitializing {
  admin = _admin;
  emit MetadataUpdated(msg.sender, metadataIpfsHash);
}
```

Example

For reference, Fee recipient invalidates `address(0)`

contracts/vaults/modules/VaultFee.sol:L36-L39

```
function _setFeeRecipient(address _feeRecipient) private {
  _checkHarvested();
  if (_feeRecipient == address(0)) revert Errors.InvalidFeeRecipient();
```

Recommendation

Consider implementing a 2-step vault admin transfer and checking that the admin is not `address(0)` to prevent any mistake.

4.11 Where Possible, a Specific Contract Type Should Be Used Rather Than Address Minor Acknowledged

Resolution

Acknowledged - By Design

Description

Declare state variables with the best type available and downcast to address if needed. Typecasting inside the corpus of a function is unneeded when the parameter's type is known beforehand. Declare the best type in function arguments and state variables. Always return the best type available instead of resorting to `address` by default.

Examples

There are more instances of this pattern, but here's a list of samples:

contracts/vaults/ethereum/EthGenesisVault.sol:L71-L75

```
{
  _poolEscrow = IPoolEscrow(poolEscrow);
  _rewardEthToken = IRewardEthToken(rewardEthToken);
}
```

contracts/vaults/ethereum/EthGenesisVault.sol:L51-L61

```
constructor(
  address _keeper,
  address _vaultsRegistry,
  address _validatorsRegistry,
  address osTokenVaultController,
  address osTokenConfig,
  address sharedMevEscrow,
  address poolEscrow,
  address rewardEthToken,
  uint256 exitingAssetsClaimDelay
)
```

contracts/vaults/modules/VaultImmutables.sol:L13-L22

```
abstract contract VaultImmutables {
  /// @custom:oz-upgrades-unsafe-allow state-variable-immutable
  address internal immutable _keeper;

  /// @custom:oz-upgrades-unsafe-allow state-variable-immutable
  address internal immutable _vaultsRegistry;

  /// @custom:oz-upgrades-unsafe-allow state-variable-immutable
  address internal immutable _validatorsRegistry;
```

contracts/vaults/modules/VaultVersion.sol:L41-L53

```
/// @inheritdoc UUPSUpgradeable
function _authorizeUpgrade(address newImplementation) internal view overri
  _checkAdmin();
  if (
    newImplementation == address(0) ||
    ERC1967Utils.getImplementation() == newImplementation || // cannot rei
    IVaultVersion(newImplementation).vaultId() != vaultId() || // vault mus
    IVaultVersion(newImplementation).version() != version() + 1 || // vaul
    !IVaultsRegistry(_vaultsRegistry).vaultImpls(newImplementation) // new
  ) {
    revert Errors.UpgradeFailed();
  }
}
```

4.12 A Malicious Adversary Could Theoretically DoS the Approval of New Validators.

Description

To mitigate the withdrawal credentials front-running vulnerability, Stakewise requires oracles to sign the validators registry's Merkle tree root when approving new validators. This ensures that if a malicious operator attempts to front-run a legitimate deposit transaction with different withdrawal credentials, the Merkle tree root of the deposit contract will change, invalidating the legitimate deposit transaction through a check in the `KeeperValidators` contract:

contracts/keeper/KeeperValidators.sol:L53-L55

```
if (_validatorsRegistry.get_deposit_root() != params.validatorsRegistryRoo
  revert Errors.InvalidValidatorsRegistryRoot();
}
```

It should be noted that this mechanism potentially opens the door to a DoS attack. Specifically, a malicious actor could theoretically disrupt validator approval by front-running legitimate deposit transactions with a deposit of at least 1 ETH into the validator registry contract. However, such an attack would likely be costly and resource-intensive to sustain over time.

4.13 Follow Ethereum Secure Coding and Style Guidelines

Description

Follow the solidity style guide. Specifically, constants should be named with all capital letters with underscores separating words. Examples: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Examples

contracts/vaults/modules/VaultEthStaking.sol:L31

```
uint256 private constant _securityDeposit = 1e9;
```

contracts/vaults/modules/VaultFee.sol:L18-L19

```
uint256 internal constant _maxFeePercent = 10_000; // @dev 100.00 %
```

contracts/vaults/modules/VaultValidators.sol:L26-L27

```
uint256 internal constant _validatorLength = 176;
```

contracts/vaults/modules/VaultVersion.sol:L26-L27

```
bytes4 private constant _initSelector = bytes4(keccak256('initialize(bytes
```

4.14 `_processTotalAssetsDelta()` Should Return Early on `totalAssetsDelta == 0`

Description

Consider taking the if-branch for `totalAssetsDelta` less than or equal zero and return early instead of consuming gas on `mulDiv` and performing accounting.

contracts/vaults/modules/VaultState.sol:L101-L116

```
/**
 * @dev Internal function for processing rewards and penalties
 * @param totalAssetsDelta The number of assets earned or lost
 */
function _processTotalAssetsDelta(int256 totalAssetsDelta) internal {
  // SLOAD to memory
  uint256 newTotalAssets = _totalAssets;
  if (totalAssetsDelta < 0) {
    // add penalty to total assets
    newTotalAssets -= uint256(-totalAssetsDelta);

    // update state
    _totalAssets = SafeCast.toUint128(newTotalAssets);
    return;
  }
```

Recommendation

The logical error can be addressed by modifying the conditional check to `if (totalAssetsDelta <= 0)`. This accommodates when the `totalAssetsDelta` parameter is 0. When it is so, neither a fee should be deducted from nor should the function proceed with the reward processing. This if-statement can simply return in such a case.

```
function _processTotalAssetsDelta(int256 totalAssetsDelta) internal {
  // SLOAD to memory
  uint256 newTotalAssets = _totalAssets;
  if (totalAssetsDelta =< 0) {
    // ...
```

4.15 Unused or Duplicate Imports

Description

Several source units contain imports for libraries or contracts that are not utilized within the codebase.

Unused imports contribute to code clutter and may confuse developers about the library’s use in the contract. Keeping the codebase clean can help with maintainability and readability.

Example

- Unused Import

contracts/vaults/modules/VaultEnterExit.sol:L8

```
import {IKeeperRewards} from '../../interfaces/IKeeperRewards.sol';
```

- Unused Import

contracts/vaults/ethereum/EthPrivVault.sol:L14

```
import {VaultVersion} from '../modules/VaultVersion.sol';
```

- Duplicate Import

contracts/vaults/ethereum/EthErc20Vault.sol:L8-L10

```
import {IEthVaultFactory} from '../../interfaces/IEthVaultFactory.sol';
import {IEthErc20Vault} from '../../interfaces/IEthErc20Vault.sol';
import {IEthVaultFactory} from '../../interfaces/IEthVaultFactory.sol';
```

Recommendation

Adhering to best practices, it is recommended to eliminate any unused imports to ensure the cleanliness of the codebase. Consequently, the removal of these unused imports from the contracts is advisable to maintain codebase integrity and enhance overall code quality.

Appendix 1 - Files in Scope

This audit covered the following files:

File Name	SHA-1 Hash
v3-core/contracts/vaults/ethereum/custom/EthFoxVault.sol	a559c0dc06d9af37c57e4bd61e109a499c80f549
v3-core/contracts/interfaces/IValidatorsRegistry.sol	948515c126bfbd5c1c54ce272ba23fbfd2e1ea4d
v3-core/contracts/interfaces/IVaultVersion.sol	96afc723c536b6fecb0e2481442fe63eaf354b64
v3-core/contracts/interfaces/IVaultEnterExit.sol	344912e106b13da951687504c590277e4d25375d
v3-core/contracts/interfaces/IVaultMev.sol	683d3e3b04ddb663a9d2e1903c956d60b6c4d989
v3-core/contracts/interfaces/IOwnMevEscrow.sol	530ceeeb1a86e9bf2ddd4c99b629abc3c320f471

File Name	SHA-1 Hash
v3-core/contracts/interfaces/IKeeper.sol	75d3628c68bf7e1e050d9c513b68e4a6255efaa8
v3-core/contracts/interfaces/IVaultBlocklist.sol	9102b95fed3d3f0a1cbb3197ed7c36806332e2a5
v3-core/contracts/interfaces/IVaultsRegistry.sol	8e47a0c2c72ff6d0e3cc2d117156ba82fac36538
v3-core/contracts/interfaces/IBalancerRateProvider.sol	9c7db0662620eb2adeedc6470f599f095f5fbe35
v3-core/contracts/interfaces/IVaultOsToken.sol	fb40e625fd05b0eb95f5d4891a4b3c95987897c3
v3-core/contracts/interfaces/IVaultToken.sol	ecbea9bf269d437e472235ef1e7cde034fcbcc9a
v3-core/contracts/interfaces/IRewardSplitter.sol	e813f82b854f3ce30e7893cdeca923f82008e5c2
v3-core/contracts/interfaces/IEthGenesisVault.sol	44041391a9bc7469cb7643c1c79fc9dacf78582b
v3-core/contracts/interfaces/IEthVault.sol	c474e8e093b3469c0c127e83203f9103fab29a63
v3-core/contracts/interfaces/IOsToken.sol	c352c8b8b5213b8043a20542d879d748f4a3362f
v3-core/contracts/interfaces/IEthValidatorsRegistry.sol	39146dd2f64598d0d5453474eeda7207cb6ffeea

File Name	SHA-1 Hash
v3-core/contracts/interfaces/ICumulativeMerkleDrop.sol	0f982f992b34243c617c997ce06ff8f551972738
v3-core/contracts/interfaces/IEthPrivVault.sol	d1de113ecad4140f16740fdb0550fa5a89fa6065
v3-core/contracts/interfaces/IVaultEthStaking.sol	0347d0ce52223fb3de6ec76218e66105cfbe50ca
v3-core/contracts/interfaces/IChainlinkAggregator.sol	be11cbe91933bdb2d55c9abfe5da31f955f2bda3
v3-core/contracts/interfaces/IKeeperValidators.sol	eb289049273af3d741dd4bbb1587df620c2f58a4
v3-core/contracts/interfaces/IEthPrivErc20Vault.sol	f3af338433f52be229e4da00db9727d782df690d
v3-core/contracts/interfaces/IRewardEthToken.sol	4caf62aaee67b06214b4948a88ec61ed82492dcf
v3-core/contracts/interfaces/IOsTokenConfig.sol	4011d66f1d0ce735b91364a3d57b89aaa3115013
v3-core/contracts/interfaces/IKeeperOracles.sol	e7238851d56892093d1819ec679aa61b5cce7e7b
v3-core/contracts/interfaces/ISharedMevEscrow.sol	a922c5cc3e1feccb2c148678a41585a19e2f5f8d
v3-core/contracts/interfaces/IEthFoxVau	ca961166b631257210fbc87f9ef402ffde01587a

File Name	SHA-1 Hash
lt.sol	
v3-core/contracts/interfaces/IVaultFee.sol	34f760b7c026c979ef9281b6221df0c5e293470b
v3-core/contracts/interfaces/IVaultAdmin.sol	cc75e80da39468bbf2b324b1f99c82cb0f046803
v3-core/contracts/interfaces/IEthErc20Vault.sol	598b955035d3ddbf48e374db39a01f0a3f9125cb
v3-core/contracts/interfaces/IPoolEscrow.sol	9761b911e6deb1a6ecc518b4c4d0c9c7b8aa36dc
v3-core/contracts/interfaces/IOsTokenVaultController.sol	a95c44197c9e758f87d5a509663e6f860613a9c1
v3-core/contracts/interfaces/IChainlinkV3Aggregator.sol	5564577ba1f87bd9a527d5f1cf01541e7caedad5
v3-core/contracts/interfaces/IRewardSplitterFactory.sol	ad6ff2ce4815f36ff354542980cbeb66c9e476e7
v3-core/contracts/interfaces/IKeeperRewards.sol	aeae2c4e0abf14cb72ba204805b5092b79d8bf6c
v3-core/contracts/interfaces/IMulticall.sol	bfb9294d62f5415e0c73fe856cd73500f1660b5e
v3-core/contracts/interfaces/IVaultValidators.sol	ec11df0dceebe7b467b52d7db76f63bd85ec88ac

File Name	SHA-1 Hash
v3-core/contracts/interfaces/IVaultState.sol	fe491d961b8e3b0e2432ed0851945154f80fbcc8
v3-core/contracts/interfaces/IEthVaultFactory.sol	086b76d40bee2d16ece182211a647639da4a2804
v3-core/contracts/interfaces/IVaultWhitelist.sol	7ea858a770409ca0b9431b53cf2140b3a84c773e
v3-core/contracts/vaults/VaultsRegistry.sol	5740a95e2c21db1f08d5059a929e4a0a161b8090
v3-core/contracts/vaults/modules/VaultMev.sol	d125688c171a575d615f60cbeabbc52fd84c4011
v3-core/contracts/vaults/modules/VaultAdmin.sol	dcd0411fb72ae2ee6423b6989376aee671905088
v3-core/contracts/vaults/modules/VaultToken.sol	fcfa29241f882f5fedd6cdb510eed45b061a62a0
v3-core/contracts/vaults/modules/VaultImmutables.sol	18eb9f5748e5cfc51cac48c8e6bf01df0f4b2105
v3-core/contracts/vaults/modules/VaultWhitelist.sol	396eb04bbc6ca9c309fb7ee1ae0c3a0202e19d30
v3-core/contracts/vaults/modules/VaultOsToken.sol	6dcd916ae3ddfdd2ba0dd861311453d4ae599fc5
v3-core/contracts/vaults/modules/Vault	abc41d2686de11f241e90d27337850d5e7600d19

File Name	SHA-1 Hash
Validators.sol	
v3-core/contracts/vaults/modules/VaultBlocklist.sol	db23cc027b8200b209df26f672c3444769666496
v3-core/contracts/vaults/modules/VaultState.sol	dc6e667b680bbbb323b11c9a5b8588b21c36dca0
v3-core/contracts/vaults/modules/VaultEnterExit.sol	ba03e8f9b29f2fcf1c37c01b017f1b4181f13fb2
v3-core/contracts/vaults/modules/VaultVersion.sol	b9e6f6f14b09b90e73b1a2bbdcc7efbc445d686e
v3-core/contracts/vaults/modules/VaultFee.sol	0ee29804186fade80fd7e2b5974e485b6653b573
v3-core/contracts/vaults/modules/VaultEthStaking.sol	30f4986ffbd6f0698c44a5f77df6e00a305d5a4b

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report 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. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute 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 absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself 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. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that

Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US

AUDITS

FUZZING

SCRIBBLE

BLOG

TOOLS

RESEARCH

ABOUT

CONTACT

Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

CAREERS

PRIVACY POLICY

Email*

e-mail address

→

POWERED BY consensys