
# **Vemo Network** Security Analysis by MoonPool

# 1. Overview Abstract

In this report, we consider the security of the Vemo Network project. Our main task is to find and describe security issues in the smart contracts of the platform to the team.

## Limitations and use of the report

Broadly speaking, the assessments can not uncover all vulnerabilities of the given smart contracts, thus, it is not guaranteed that the system is secured even if no vulnerabilities are found. The focus of the assessments was limited to given smart contracts, other contracts were excluded (including external libraries or third party codes).

## Audit Summary

We have found **2** high severity issues, **4** medium severity issues and **8** low severity issues in the initial report with the commit hash bd50a6a03f5ff14a0d8ec1b39b18c0f85f52a098 .

After the initial report, the team has made some changes to the new commit hash dd1efbb52da3b2c50ebabfc13b2fe6d83a580dd9. In the second report with the new commit hash, we have found **0** high severity issue, **2** medium severity issues and **3** low severity issues.

After the second report, the team has made some changes to the new commit hash 25c34cdbf6cf6c6f72469eaeb658780b98a12536 to fix reported issues.

## Recommendations

We recommend the team to fix all issues, as well as test coverage to ensure the security of the contracts.

# 2. Assessment Overview

## Scope of the audit

Source codes for the audit was initially taken from the commit hash *bd50a6a03f5ff14a0d8ec1b39b18c0f85f52a098*. In-scope contracts for audit:

- src/WalletFactory.sol
- src/terms/VePendleTerm.sol
- src/accounts/AccountV3.sol
- src/accounts/NFTAccountDelegable.sol
- src/lib/IterableMap.sol
- src/CollectionDeployer.sol
- src/helpers/VemoDelegationCollection.sol
- src/interfaces/ICollectionDeployer.sol
- src/interfaces/IDelegationCollection.sol
- src/interfaces/IExecutionTerm.sol
- src/interfaces/IWalletFactory.sol

The second audit iteration was taken from the commit hash dd1efbb52da3b2c50ebabfc13b2fe6d83a580dd9 with similar in-scope contracts for audit.

The final report was taken from the commit hash 25c34cdbf6cf6c6f72469eaeb658780b98a12536 with similar in-scope contracts for audit

Other source codes are out of scope for this report.

# 3. System Overview

The system overview is based on the provided documentation that is available on their website here.

## 1. Introduction

**Vemo Network** is a complete and standalone smart contract system by leveraging ERC-721 and ERC-6551 standards, Vemo tokenizes locked positions into **NFT Accounts** or **Smart Vouchers**. These are referred to as independent asset layers, enabling trading, lending, and fundraising in the market. **For this audit, we're only requesting the audit for NFT Account.**

## 2. Expectations on How It Works

Vemo strictly followed the design of EIP6551, each TBA is permanently associated with an ERC721 NFT. User can use TBA with wallet connect as normal wallet. TBA implementation is only upgrade by their owner - who owns the associated NFT. The tokenURI is generated by an onchain descriptor - which mean you can have the metadata by calling `tokenURI(tokenID)` . Each Dapps/Partner can build their own descriptor.

Vemo adheres strictly to the EIP-6551 design, ensuring that each Token Bound Account (TBA) is permanently linked to an ERC-721 NFT. Users can interact with their TBA using WalletConnect, just like a regular wallet. The TBA's implementation is upgradable solely by the NFT owner, ensuring exclusive control. The metadata is generated on-chain through a descriptor, accessible by calling `tokenURI(tokenID)`, and allowing dApps or partners to develop custom descriptors.

# 3. Protocol Overview

- **WalletFactory**: Acts as a gateway for any dApp to interact with and create both TBA collections and delegation collections. It facilitates the deployment and management of collections, streamlining their interaction within the ecosystem.
- **Registry**: This is a singleton contract serving as the main access point for all TBA address queries. Utilizing the CREATE2 opcode, it allows for the deterministic creation and retrieval of TBAs linked to NFTs. The Registry is central to managing and querying the existence of TBAs, ensuring consistent and unique account creation for each NFT.
- **CollectionDeployer**: Responsible for deploying TBA collections and delegation collections. It ensures that each collection is properly established and linked to the relevant NFTs, maintaining the integrity of the TBA structure.
- **Token Bound Account (TBA)**: The Vemo protocol's core feature allows an ERC-721 token (NFT) to own assets and execute transactions through a TBA. This advancement provides NFTs with their own on-chain accounts, enabling interaction with smart contracts, token holding, and autonomous actions. Each TBA is permanently associated with a single NFT, with control vested in the NFT holder, who can use WalletConnect for app access. Vemo offers various TBA types based on EIP-6551, focusing mainly on NFTAccountDelegable and AccountV3.sol in this scope.
- **Collection**: Vemo supports two collection types—TBA collections and delegation collections. TBA collections are AutoURI collections created either by dApps, partners, or the Vemo team, and are managed through the WalletFactory. Delegation collections are associated with specific terms, such as rules, whitelisted contracts, and actions tailored to specific business logic or dApps. This setup allows users to delegate tasks like voting and yield harvesting to others while retaining full control over TBA assets.
- **Term**: A predefined set of rules and actions for specific protocols, maintained by the Vemo team. Each term can be linked to multiple collections, defining the scope of permissible actions within a protocol. This structure enhances the delegation process by clearly defining the rules under which an individual can interact with a TBA.

Documentations to explain more details about expectation of each contract's implementation were not provided.

# 4. Findings - Initial Report

We have found 2 high severity issues, 4 medium severity issues and 8 low severity issues.
Some issues have been addressed, while others are acknowledged.

## High severity issues

**The audit showed 2 high severity issues.**

### [Code Changed] 4.1 VePendleTerm.sol: Can not transfer tokens out of the term contract due to revert in transferFrom

```
if (_rewardAssets[i] != address(0)) {
    IERC20(_rewardAssets[i]).transferFrom(address(this), _owner, amountOwner);
    IERC20(_rewardAssets[i]).transferFrom(address(this), _farmer, amountFarmer);
}
```

**Issue:** In the ***VePendleTerm.sol:split***, when reward asset is a token (not native), it calls *transferFrom* to move the reward asset from the contract itself to the *_owner* and *_farmer*. In order to use *transferFrom*, it requires the *from* to approve token allowance to the *caller*, in this case, both *from* and *caller* are address(this) and there is no logic to give token allowance from address(this) - i.e the term contract - to itself, thus, these 2 calls will be always reverted.
**Recommendation:** We recommend to use a normal *transfer* instead of *transferFrom*. Best practice is to use *safeTransfer* from ***SafeERC20*** which is already imported but unused.
**Code Changed**: Split logic in the **VePendleTerm.sol** has been removed.

## [Code Changed] 4.2 NFTAccountDelegable.sol: double-paying farmer in harvest and distribute rewards, potential revert due to distributing over total reward amount

```
for (uint i = 0; i < rewardTokens.length; i++) {
    rewards[i] = _balanceOf(rewardTokens[i]) - rewards[i];
    if (rewards[i] == 0) continue;

    if (rewardTokens[i] == address(0)) {
        farmer.call{value: rewards[i] * splitRatio / 10_000}("");
    } else {
        IERC20(rewardTokens[i]).transfer(farmer, rewards[i] * splitRatio / 10_000);
    }
}
```

**Issue:** In the **NFTAccountDelegable.sol:_harvestAndDistributeReward**: after harvesting rewards, it already distribute part of the reward to the *farmer* based on the **splitRatio**. Then it passes the whole reward amounts to the term contract to split the rewards between the farmer and owner.

```
function split(
    address payable _owner,
    address payable _farmer,
    uint256[] memory rewards
) public {
    for (uint i = 0; i < _rewardAssets.length; i++) {
        uint256 amountFarmer = (rewards[i] * splitRatio) / 10000;
        uint256 amountOwner = rewards[i] - amountFarmer;

        // avoid reentrancy
        rewards[i] = 0;

        if (_rewardAssets[i] != address(0)) {
            IERC20(_rewardAssets[i]).transferFrom(address(this), _owner, amountOwner);
            IERC20(_rewardAssets[i]).transferFrom(address(this), _farmer, amountFarmer);
        } else {
            _owner.call{value: amountOwner}("");
            _farmer.call{value: amountFarmer}("");
        }
    }
}
```

In the **VePendleTerm:split**, it again uses the original reward amounts to calculates the share to the *farmer* and *owner* based on the **splitRatio**, it means it is double-paying the *farmer*.

On the other hand, the total of amounts it transfers to the owner and farmer exceeds the total reward amount it has been received, potentially revert the harvest and distribute reward process as insufficient rewards to pay.

Example:

- Assume: 1 reward asset, rewards[0] = 100, splitRatio = 100 (1%).
- In *_harvestAndDistributeReward*, it transfers rewards[0] * splitRatio / 10_1000 = 1 to the farmer.
- It calls term.split(owner, farmer, rewards[0]).
- In *_split*, it calculates:
  - amountFarmer = rewards[0] * 100 / 10000 = 1.
  - amountOwner = rewards[0] - 1 = 99.
  - It transfers 1 to the farmer, and 99 to the owner.

Thus, in total, it transfers 101 to the farmer and owner, even though the total reward amount is 100.

**Recommendation:** Double-paying should be mitigated.

**Code Changed**: Split logic in the **VePendleTerm.sol** has been removed.

# Medium severity issues

**The audit showed 4 medium severity issues.**

## [Code Changed] 4.3 VePendleTerm.sol: anyone can call split function to transfer all tokens out

```
function split(
    address payable _owner,
    address payable _farmer,
    uint256[] memory rewards
) public {
    for (uint i = 0; i < _rewardAssets.length; i++) {
        uint256 amountFarmer = (rewards[i] * splitRatio) / 10000;
        uint256 amountOwner = rewards[i] - amountFarmer;

        // avoid reentrancy
        rewards[i] = 0;

        if (_rewardAssets[i] != address(0)) {
            IERC20(_rewardAssets[i]).transferFrom(address(this), _owner, amountOwner);
            IERC20(_rewardAssets[i]).transferFrom(address(this), _farmer, amountFarmer);
        } else {
            _owner.call{value: amountOwner}("");
            _farmer.call{value: amountFarmer}("");
        }
    }
}
```

**Issue:** In the **VePendleTerm:split**, anyone can call this function with any values for _owner, _farmer, rewards, thus, if there is any reward assets in the contract, it could be drained immediately by anyone. The “avoid reentrancy” doesn’t work in this function as well.

**Recommendation:** Should have specific roles that can call this function.

**Code Changed**: Split logic in the **VePendleTerm.sol** has been removed.

## [Acknowledged] 4.4 Design - owners must fully trust the Term’s owner with all their assets.

**Issue:** As the definition, the Term contract contains predefined set of rules and actions for specific protocols, maintained by the Vemo team.

In the **VePendleTerm.sol:canExecute**, it will be passed if both *whitelist* and *selectors* are empty, which means the *delegate* can execute an arbitrary transaction. The *whitelist* and *selectors* can be updated by the Term’s owner any time without any time lock.

The **VePendleTerm** contract is designed to be upgradable by the owner without any time lock, thus, it could be upgraded to bypass all checks and execute an arbitrary transaction.

From above design, the NFTAccount owner must fully trust the Term’s owner to not:

- maliciously update *whitelist* and *selectors*.
- upgrade Term’s contract to bypass all checks.

Please note that when it allows to execute an arbitrary transaction, it could potentially drain all funds from the NFTAccount owner, not just pre-defined assets.

**Comment from team**: Team acknowledged the problem and is considering implementing two steps to change any term configuration.

## [Acknowledged] 4.5 Design - Account owner as the root token owner, can execute any transactions and give no share to the delegation

**Issue:** In the **NFTAccountDelegable**, it appears that the **revoke** logic is with a revoke timeout, possibly to prevent the owner from revoking *delegation* immediately. This helps to give more trusts to the *delegation*, like giving them enough time to harvest rewards and take their shares.

However, as the root token owner, the owner can still execute an arbitrary transaction immediately, thus, the owner can always harvest rewards and transfer rewards out without giving any shares to the *delegation*.

In general, the *delegation* still need to full trusts the account owner to give share, and may affect the overall design of the protocol.

**Acknowledged**: Team has acknowledged this issue but only apply in the next version.

### [No issue] 4.6 WalletFactory.sol: attacker can create an account with any implementation in advance

**Issue:** In the **WalletFactory.sol:_createAndInitializeTBA**, it doesn't revert if the account has been created and return the account address. This function is also assuming this logic **IAccountProxy(account).initialize(walletImpl)** will update the wallet implementation to the **walletImpl**. However, anyone can create an account from **AccountRegistry** and initialize any implementation, thus, an attacker can create an account in advance with malicious implementation and later use **WalletFactory** to pretend creating a valid TBA

**No Change**: Team confirms that an implementation can not be deployed with the verification and approve from Vemo team thru AccountGuardian.

## Low severity issues

**The audit showed 8 low severity issues.**

### [Acknowledged] 4.7 WalletFactory.sol: redundant use of name and symbol in hashKey in _deployNFTCollection

**Issue:** In **WalletFactory.sol: _deployNFTCollection**, it uses (nam, symbol, salt) to calculate **hashKey** to avoid collision, however, the **VemoWalletCollection** is deployed with salt, thus, it is still reverted if using different name/symbol but the same salt, thus, we can reduce to use only *salt* as the **hashKey**.

**Recommendation:** can use only *salt* as the **hashKey**.

**Acknowledged**: Team has acknowledged this issue but no changes.

### [Acknowledged] 4.8 WalletFactory.sol: unused variable should be removed

**Issue:** In **WalletFactory.sol**, the *dappURIs* is commented as unused and should be removed.

Similarly, the *withdrawalFeeBps* is also unused.

**Recommendation**: unused variable should be removed.

**Acknowledged**: Team has acknowledged this issue but no changes.

## [Code Changed] 4.9 WalletFactory.sol: move all defined variables in the same place

**Issue:** As **WalletFactory.sol** is upgradable, to avoid storage collision when upgrading, we recommend to move all defined variables in the same place so that it is easier to manage. Currently *feeReceiver*, *depositFeeBps* and *withdrawalFeeBps* are defined in the middle of the codebase and should be move on top.

**Recommendation**: move *feeReceiver*, *depositFeeBps* and *withdrawalFeeBps* on top for easier managing.

**Code Changed**: Variables have been moved to correct place.

## [Acknowledged] 4.10 WalletFactory.sol: deposit logics seems redundant as owner can directly transfer tokens to the walletAddress

**Issue:** The owner can directly transfer native and tokens to the walletAddress instead of using deposit functions (with deposit fees), thus, it is redundant to have these functions in the contract.

**Recommendation**: these functions and associated variables should be removed.

**Acknowledged**: Team has acknowledged this issue but no changes.

## [Acknowledged] 4.11 VePendleTerm.sol: unused variable should be removed

**Issue:** *guardian* is defined but seems unused.

**Recommendation**: unused variable should be removed.

**Acknowledged**: Team has acknowledged this issue but no changes.

## [Acknowledged] 4.12 IterableMap.sol: Potential Accumulation of Zero-Value Entries

**Issue:** The linked list can accumulate entries with zero values over time because the `set` function does not remove entries from the linked list when their values are set to zero. This can lead to inefficient iterations and increased storage usage.
**Recommendation**: Modify the `set` function to remove entries from the linked list when their value is set to zero, or ensure that the `compact` function is regularly called to clean up zero-value entries.
**Acknowledged**: Team has acknowledged this issue but no changes.

## [Code Changed] 4.13 CollectionDeployer.sol: Revert if deploying an address with the same salt, thus, checking for ownership is redundant

**Issue:** This check *(Ownable(collection).owner() != address(this)) revert IssuedByOtherFactory()* is redundant as if the *collection* has been deployed, calling *new VemoDelegationCollection{salt: keccak256(abi.encode(term, issuer))}()* should already revert.
**Recommendation**: The check for ownership can be removed.
**Code Changed**: Redundant check has been removed.

## [Code Changed] 4.14 VemoDelegationCollection.sol: High gas consumption in constructor as fetching parameters multiple times

**Issue:** In the constructor, it fetches **parameters** from the deployer 4 times with 6 storage parameters.
**Recommendation**: Should have separate functions to retrieve params to save gas.
**Code Changed**: Separated functions to retrieve corresponding params have been implemented.

# 5. Findings - Second Iteration

Apart from acknowledged issues in the initial reports, in the second audit iteration with the updated code, we have found 2 medium severity issues and 3 low severity issues.

Some issues have been addressed, while others are acknowledged.

## Medium severity issues

**The audit showed 2 medium severity issues.**

### [Code Changed] 5.1 VeTerm.sol: inconsistent type data for action in disallowAction function, potentially unable to disable majority actions

**Issue: allowedActions** and **harvestingActions** are mapping from ***bytes24 => bool*** with the expectation that action is of ***bytes24*** type. However, in the **disallowAction** function, it only disables action of type ***bytes4,*** means actions with values outside of ***bytes4*** can not be disabled.

```
function disallowAction(ACTION_TYPE _type, bytes4 action) public onlyOwner {
    if (_type == ACTION_TYPE.EXECUTION) {
        allowedActions[action] = false;
    } else {
        harvestingActions[action] = false;
    }
}
```

**Suggestion**: **action** param in the **disallowAction** function should be of type ***bytes24***.
**Code changed**: **action** has been changed to **bytes24** type

### [Acknowledged] 5.2 VemoDelegationCollection: isValidDelegatee always returns true for incorrect tokenId

**Issue:** the **isValidDelegatee(tokenId)** is returning true if **revokingRoles[tokenId] == 0**, where this condition is true for all invalid **tokenId**, including none existing ***tokenId*** (not created yet, or already burnt).

**Suggestion:** With the nature of the function, it should return **false** if the ***tokenId*** has no owner.

**Comment from team:** The **isValidDelegatee** function is used 1 time for now, before calling the code has already checked the ownership of tokenID (this check includes the existence)

## Low severity issues

**The audit showed 3 low severity issues.**

### [Code Changed] 5.3 VeTerm.sol: unused import console.sol

**Issue:** ***console.s***ol is imported for the purpose of debugging, but it should be removed in the actual implementation.

**Suggestion:** remove ***console.sol*** import.

**Code changed**: unused ***console.sol*** import has been removed.

### [Acknowledged] 5.4 VemoDelegationCollection: redundant get both name and symbol in the _ERC721Params

**Issue:** The **_ERC721Params** only gets either **name** or **symbol** info, thus, it is redundant to always get both values

**Suggestion:** It should check if the index is 0 or 1 and get the corresponding info, so only 1 call is needed in any cases.

**Comment from team:** None

## [Code Changed] 5.5 NFTAccountDelegable: compares tokenContract with collection.issuer when validating signature

**Issue:** In the **NFTAccountDelegable.sol:_isValidSignature** function, it gets the **issuer** from the delegation collection, to compares with the **issuer** which is returned from **ERC6551AccountLib.token()**. However, the **ERC6551AccountLib.token()** returns 3 params as (*chainID*, ***tokenContract***, *tokenId*), i.e ***tokenContract*** here is considered as **issuer**.

**Suggestion:** Please make sure this logic is expected, should have a better name or explanation with the logic.

```
// extract delegation from signature
address collection = BytesLib.toAddress(signature, 65);
if (!isDelegate(collection)) revert UnknownCollection();

address signer;
ECDSA.RecoverError _error;
(signer, _error,) = ECDSA.tryRecover(hash, BytesLib.slice(signature, 0, 65));

if (_error != ECDSA.RecoverError.NoError) return false;

(, address issuer, uint256 tokenId) = ERC6551AccountLib.token();

require(IERC721(collection).ownerOf(tokenId) == signer, "!delegate");
require(IDelegationCollection(collection).issuer() == issuer, "!issuer");

address term = IDelegationCollection(collection).term();
return IExecutionTerm(term).isValidSignature(hash, signature);
```

**Code changed:** team has renamed some variables to make it clearer.

# 6. Design Comment

The current contract design relies heavily on trust in off-chain operations, especially when updating actions in the term contract without asking for user approval. This creates a risk, as users need to fully trust that the off-chain actions are done correctly and in their favor. To make this safer, we suggest adding a Time Lock mechanism for any important updates that might affect user assets in the account contracts. A Time Lock would give users extra time to review the changes and, if necessary, withdraw their assets before the updates take place. This would increase transparency and allow users to verify the changes before they are finalized. By adding this feature, the system can reduce the amount of trust required from users and improve the contract's security. We also encourage users to be cautious and verify off-chain data and updates to protect their assets.

# 7. Testing

The team has provided tests for all contracts, they do not provide 100% full coverage yet. We strongly recommend the team to add full test coverage to ensure the security of the codes.