SuperDeed V.2

Smart Contract Audit Report
Prepared for SuperLauncher

Date Issued: Jan 19, 2024
Project ID: AUDIT2022002
Version: v3.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022002
Version	v3.0
Client	SuperLauncher
Project	SuperDeed V.2
Auditor(s)	Weerawat Pawanawiwat Suvicha Buakhom Peeraphut Punsuwan Darunphop Pengkumta
Author(s)	Darunphop Pengkumta
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
3.0	Jan 19, 2024	Update logo	Peeraphut Punsuwan
2.0	Jan 18, 2024	Update chain to Zksync	Peeraphut Punsuwan
1.0	Jan 18, 2022	Full report	Darunphop Pengkumta

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by SuperLauncher, Inspex team conducted an audit to verify the security posture of the SuperDeed V.2 smart contracts between Jan 11, 2022 and Jan 12, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of SuperDeed V.2 smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 medium, 1 low, 1 very low, and 1 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved in the reassessment. Therefore, Inspex trusts that SuperDeed V.2 smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

SuperLauncher is an investment DAO that funds and collaborates with projects that are shaping the future of the blockchain. SuperLauncher facilitates mass participation in Seed/Private/IDO rounds through its smart contract architecture and offers a feature-rich platform that powers flexible and decentralized management of capital.

SuperDeed V.2 is a claimable NFT. The users who hold the SuperDeed V.2 NFT entitlement can claim a token in the future releases as configured by the project owner. The entitlement supports the releases of tokens with ERC20, ERC1155, and ERC721 standards. Each campaign can be configured with multiple groups of vesting releases for the users to claim. Each group can be configured with different types of vesting.

Scope Information:

Project Name	SuperDeed V.2
Website	https://superlauncher.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Zksync
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 11, 2022 - Jan 12, 2022
Reassessment Date	Jan 17, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: cdb5007eac5938b0faed785c60545b2d4889ba17)

Contract	Location (URL)
DataStore	https://github.com/SuperLauncher/v2-superdeed/blob/cdb5007eac/contracts/core/DataStore.sol
Factory	https://github.com/SuperLauncher/v2-superdeed/blob/cdb5007eac/contracts/Factory.sol
Manager	https://github.com/SuperLauncher/v2-superdeed/blob/cdb5007eac/contracts/Manager.sol
RolesRegistery	https://github.com/SuperLauncher/v2-superdeed/blob/cdb5007eac/contracts/RolesRegistry.sol
SuperDeedV2	https://github.com/SuperLauncher/v2-superdeed/blob/cdb5007eac/contracts/SuperDeedV2.sol

Reassessment: (Commit: fd70fe63fefedf4bf8433fcd4b202a05bbf07044)

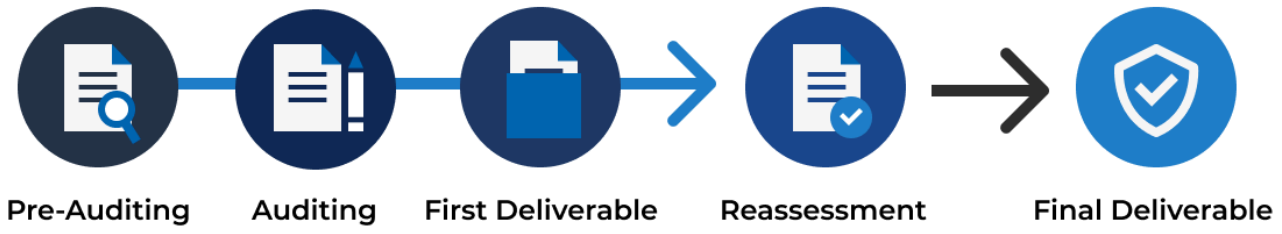
Contract	Location (URL)
DataStore	https://github.com/SuperLauncher/v2-superdeed/blob/fd70fe63fe/contracts/core/DataStore.sol
Factory	https://github.com/SuperLauncher/v2-superdeed/blob/fd70fe63fe/contracts/Factory.sol
Manager	https://github.com/SuperLauncher/v2-superdeed/blob/fd70fe63fe/contracts/Manager.sol
RolesRegistery	https://github.com/SuperLauncher/v2-superdeed/blob/fd70fe63fe/contracts/RolesRegistery.sol
SuperDeedV2	https://github.com/SuperLauncher/v2-superdeed/blob/fd70fe63fe/contracts/SuperDeedV2.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Improper Kill-Switch Mechanism

Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

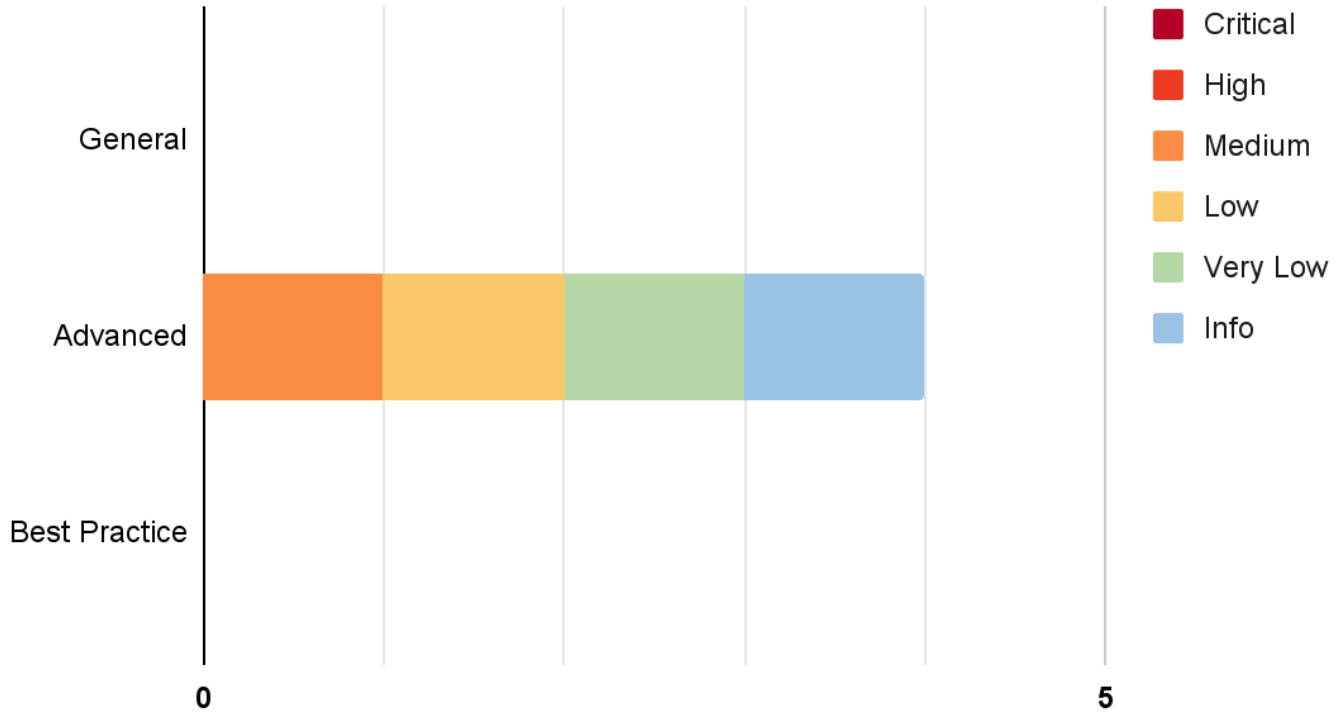
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	**Very Low**	**Low**	**Medium**
Medium	**Low**	**Medium**	**High**
High	**Medium**	**High**	**Critical**

4. Summary of Findings

From the assessments, Inspex has found 4 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Improper Setting Asset Details	Advanced	**Medium**	Resolved
IDX-002	Improper DAO Privilege in Emergency Withdrawal Process	Advanced	**Low**	Resolved
IDX-003	Modification of Vesting Information After Finalization	Advanced	**Very Low**	Resolved
IDX-004	Starting of Vesting Without Funded Group	Advanced	**Info**	Resolved

* The mitigations or clarifications by SuperLauncher can be found in Chapter 5.

5. Detailed Findings Information

5.1. Improper Setting of Asset Details

ID	IDX-001
Target	SuperDeedV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The DAO can transfer out the fund after the project owner has funded the token. Moreover, when the token address has been changed, the users cannot claim the token after the vesting has started. **Likelihood:** Low Only the project owner or the configurator can set the asset details. Moreover, the emergency withdrawal action requires the multisig mechanism to execute the emergency withdrawal function.
Status	Resolved SuperLauncher team has resolved this issue as suggested by adding a validation on the `_isFunded` state on the `setAssetDetails()` function to prevent the asset details from being set after being funded in commit `fd70fe63fefedf4bf8433fcd4b202a05bbf07044`.

5.1.1. Description

The project owner can fund the token into the contract by executing the `fundInForGroup()` function.

SuperDeedV2.sol

```
77  function fundInForGroup(uint groupId, string memory groupName, uint
    tokenAmount) external notLive onlyProjectOwnerOrApprover {
78      _check(groupId, groupName);
79      _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Invalid
    address");
80
81      // Check required token Amount is correct?
82      DataType.Group storage group = _groups().items[groupId];
83      _require(tokenAmount == group.info.totalEntitlement, "Wrong token amount");
84
85      // Group must be finalized and not yet fund in
86      _require(group.state.finalized, "Not yet finalized");
87      _require(!group.state.funded, "Already funded");
88      group.state.funded = true;
```

```
89
90      DataType.AssetType assetType = _asset().tokenType;
91      if (assetType == DataType.AssetType.ERC20) {
92          IERC20(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), tokenAmount);
93      } else if (assetType == DataType.AssetType.ERC1155) {
94          IERC1155(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), _asset().tokenId, tokenAmount, "");
95      } else {
96          // Verify that the amount has been deposied already ?
97          DataType.Erc721Handler storage handler = _store().erc721Handler;
98
99          uint totalDeposited721 = handler.erc721IdArray.length;
100         _require(totalDeposited721 >= (handler.numUsedByVerifiedGroups +
    tokenAmount), "Insufficient deposited erc721");
101         handler.numUsedByVerifiedGroups += tokenAmount;
102     }
103     emit FundInForGroup(msg.sender, groupId, groupName, tokenAmount);
104     _recordHistory(DataType.ActionType.FundInForGroup, groupId, tokenAmount);
105 }
```

After the project owner has funded, the `configurator` role can also set new asset details by executing the `setAssetDetails()` function to change the token asset to another token.

SuperDeedV2.sol

```
65  function setAssetDetails(address tokenAddress, DataType.AssetType tokenType,
    uint tokenIdFor1155) external notLive onlyProjectOwnerOrConfigurator {
66      _setAssetDetails(tokenAddress, tokenType, tokenIdFor1155);
67      _recordHistory(DataType.ActionType.SetAssetAddress, uint160(tokenAddress),
    uint(tokenType));
68  }
```

The DAO can execute the `daoMultiSigEmergencyWithdraw()` function to transfer out the ERC20 token funded in, when the token is not the asset to be distributed.

SuperDeedV2.sol

```
309 function daoMultiSigEmergencyWithdraw(address tokenAddress, address to, uint
    amount) external override onlyDaoMultiSig {
310
311     // If withdrawn token is the asset, then we will require projectOwner to
    approve.
312     // Every approval allow 1 time withdraw only.
313     if (tokenAddress == _asset().tokenAddress) {
314         _require((amount <= _emergencyMaxAmount) && (block.timestamp <=
    _emergencyExpiryTime), "Criteria not met");
315
```

```
316         // Reset
317         _emergencyMaxAmount = 0;
318         _emergencyExpiryTime = 0;
319
320         _transferAssetOut(to, amount);
321     } else {
322         // Withdraw non asset ERC20
323         _transferOutErc20(tokenAddress, to, amount);
324     }
325     emit DaoMultiSigEmergencyWithdraw(to, tokenAddress, amount);
326 }
```

Moreover, The users cannot claim the funded token because the token asset has changed, so the transaction will be reverted in the `_transferAssetOut()` function.

DataStore.sol

```
139 function _transferAssetOut(address to, uint amount) internal {
140
141     DataType.AssetType assetType = _asset().tokenType;
142     address token = _asset().tokenAddress;
143
144     if (assetType == DataType.AssetType.ERC20) {
145         IERC20(token).safeTransfer(to, amount);
146     } else if (assetType == DataType.AssetType.ERC1155) {
147         IERC1155(token).safeTransferFrom(address(this), to, _asset().tokenId,
    amount, "");
148     } else if (assetType == DataType.AssetType.ERC721) {
149
150         DataType.Erc721Handler storage handler = _store().erc721Handler;
151         uint len = handler.erc721IdArray.length;
152         require(handler.numErc721TransferedOut + amount <= len, "Exceeded
    Amount");
153
154         for (uint n=0; n<amount; n++) {
155             uint id = handler.erc721IdArray[handler.erc721NextClaimIndex++];
156             IERC721(token).safeTransferFrom(address(this), to, id);
157         }
158         handler.numErc721TransferedOut += amount;
159     }
160 }
```

5.1.2. Remediation

Inspex suggests preventing the asset details from being modified after funding in. For example, this can be done by creating the `_isFunded` state variable in the `DataStore` contract.

DataStore.sol

```
27 uint internal _emergencyMaxAmount;
28 uint internal _emergencyExpiryTime;
29 bool internal _isFunded;
```

The `_isFunded` state should be set as true at the `fundInForGroup()` and `fundInForGroupOverride()` functions in line 89 and 120.

SuperDeedV2.sol

```
77  function fundInForGroup(uint groupId, string memory groupName, uint
    tokenAmount) external notLive onlyProjectOwnerOrApprover {
78      _check(groupId, groupName);
79      _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Invalid
    address");
80
81      // Check required token Amount is correct?
82      DataType.Group storage group = _groups().items[groupId];
83      _require(tokenAmount == group.info.totalEntitlement, "Wrong token amount");
84
85      // Group must be finalized and not yet fund in
86      _require(group.state.finalized, "Not yet finalized");
87      _require(!group.state.funded, "Already funded");
88      group.state.funded = true;
89      _isFunded = true;
90
91      DataType.AssetType assetType = _asset().tokenType;
92      if (assetType == DataType.AssetType.ERC20) {
93          IERC20(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), tokenAmount);
94      } else if (assetType == DataType.AssetType.ERC1155) {
95          IERC1155(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), _asset().tokenId, tokenAmount, "");
96      } else {
97          // Verify that the amount has been deposied already ?
98          DataType.Erc721Handler storage handler = _store().erc721Handler;
99
100         uint totalDeposited721 = handler.erc721IdArray.length;
101         _require(totalDeposited721 >= (handler.numUsedByVerifiedGroups +
    tokenAmount), "Insufficient deposited erc721");
102         handler.numUsedByVerifiedGroups += tokenAmount;
103     }
104     emit FundInForGroup(msg.sender, groupId, groupName, tokenAmount);
105     _recordHistory(DataType.ActionType.FundInForGroup, groupId, tokenAmount);
106 }
107
108
```

```
109 function fundInForGroupOverride(uint groupId, string memory groupName) external
    notLive onlyDaoMultiSig {
110
111     _check(groupId, groupName);
112     _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Invalid
    address");
113
114     // Check required token Amount is correct?
115     DataType.Group storage group = _groups().items[groupId];
116     // Group must be finalized and not yet fund in
117     _require(group.state.finalized, "Not yet finalized");
118     _require(!group.state.funded, "Already funded");
119     group.state.funded = true;
120     _isFunded = true;
121
122     emit FundInForGroupOverrided(msg.sender, groupId, groupName);
123     _recordHistory(DataType.ActionType.FundInForGroupOverrided, groupId, 0);
124 }
```

The condition to prevent the setting of new asset detail after the owner funded the token should be added in the `setAssetDetails()` function as in line 66.

SuperDeedV2.sol

```
65 function setAssetDetails(address tokenAddress, DataType.AssetType tokenType,
    uint tokenIdFor1155) external notLive onlyProjectOwnerOrConfigurator {
66     require(!_isFunded, "The asset has been funded in");
67     _setAssetDetails(tokenAddress, tokenType, tokenIdFor1155);
68     _recordHistory(DataType.ActionType.SetAssetAddress, uint160(tokenAddress),
    uint(tokenType));
69 }
```

5.2. Improper DAO Privilege in Emergency Withdrawal Process

ID	IDX-002
Target	SuperDeedV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-269: Improper Privilege Management
Risk	**Severity:** Low **Impact:** Medium The DAO can transfer the token to any arbitrary address, stealing the token from the platform owner. However, the amount is limited by the value specified on the approval by the contract owner. **Likelihood:** Low The attack can be profitable for the DAO, but it is unlikely for the majority of the DAO multisig key holders to agree on performing the malicious action.
Status	Resolved SuperLauncher team has resolved this issue as suggested by adding a `destination` parameter on the `approveEmergencyAssetWithdraw()` function to pre-define the destination address of the emergency withdrawal being used by `DAOMultisig` on the `daoMultiSigEmergencyWithdraw()` function in commit `fd70fe63fefedf4bf8433fcd4b202a05bbf07044`.

5.2.1. Description

The project owner can withdraw their token in the emergency case by requesting passing in the `maxAmount` value to the `approveEmergencyAssetWithdraw()` function. After the project owner has requested to withdraw the token, the DAO can call the `daoMultiSigEmergencyWithdraw()` function to withdraw the token to any address no matter if the address is the project owner or not.

SuperDeedV2.sol

```
303 function approveEmergencyAssetWithdraw(uint maxAmount) external override
    onlyProjectOwner {
304     _emergencyMaxAmount = maxAmount;
305     _emergencyExpiryTime = block.timestamp + Constant.EMERGENCY_WINDOW;
306     emit ApprovedEmergencyWithdraw(msg.sender, _emergencyMaxAmount,
    _emergencyExpiryTime);
307 }
308
309 function daoMultiSigEmergencyWithdraw(address tokenAddress, address to, uint
    amount) external override onlyDaoMultiSig {
310
```

```
311     // If withdrawn token is the asset, then we will require projectOwner to
    approve.
312     // Every approval allow 1 time withdraw only.
313     if (tokenAddress == _asset().tokenAddress) {
314         _require((amount <= _emergencyMaxAmount) && (block.timestamp <=
    _emergencyExpiryTime), "Criteria not met");
315
316         // Reset
317         _emergencyMaxAmount = 0;
318         _emergencyExpiryTime = 0;
319
320         _transferAssetOut(to, amount);
321     } else {
322         // Withdraw non asset ERC20
323         _transferOutErc20(tokenAddress, to, amount);
324     }
325     emit DaoMultiSigEmergencyWithdraw(to, tokenAddress, amount);
326 }
```

5.2.2. Remediation

Inspex suggests letting the project owner define the withdraw destination address to prevent the DAO from withdrawing to any address, for example:

The state to store the emergency withdrawal destination should be added to the `DataStore` contract as shown in line 29.

DataStore.sol

```
15 contract DataStore {
16
17     using SafeERC20 for IERC20;
18     using Groups for *;
19     using MerkleClaims for *;
20
21     DataType.Store private _dataStore;
22     IRoleAccess private _roles;
23
24     address public projectOwner;
25
26     // Emergency Withdrawal Support
27     uint internal _emergencyMaxAmount;
28     uint internal _emergencyExpiryTime;
29     address internal _emergencyDestination;
```

The destination address should be accepted as a parameter in the `approveEmergencyAssetWithdraw()` function and set it to the state defined as shown in line 303 and 306. The event in line 307 should also be adjusted.

SuperDeedV2.sol

```
302 // Implements IEmergency
303 function approveEmergencyAssetWithdraw(uint maxAmount, address destination)
    external override onlyProjectOwner {
304     _emergencyMaxAmount = maxAmount;
305     _emergencyExpiryTime = block.timestamp + Constant.EMERGENCY_WINDOW;
306     _emergencyDestination = destination;
307     emit ApprovedEmergencyWithdraw(msg.sender, _emergencyMaxAmount,
    _emergencyExpiryTime, _emergencyDestination);
308 }
```

The destination address should be checked in the `daoMultiSigEmergencyWithdraw()` function as shown in line 315, and reset in line 320.

SuperDeedV2.sol

```
309 function daoMultiSigEmergencyWithdraw(address tokenAddress, address to, uint
    amount) external override onlyDaoMultiSig {
310
311     // If withdrawn token is the asset, then we will require projectOwner to
    approve.
312     // Every approval allow 1 time withdraw only.
313     if (tokenAddress == _asset().tokenAddress) {
314         _require((amount <= _emergencyMaxAmount) && (block.timestamp <=
    _emergencyExpiryTime), "Criteria not met");
315         _require(to == _emergencyDestination, "Wrong withdrawal destination");
316
317         // Reset
318         _emergencyMaxAmount = 0;
319         _emergencyExpiryTime = 0;
320         _emergencyDestination = Constant.ZERO_ADDRESS;
321
322         _transferAssetOut(to, amount);
323     } else {
324         // Withdraw non asset ERC20
325         _transferOutErc20(tokenAddress, to, amount);
326     }
327     emit DaoMultiSigEmergencyWithdraw(to, tokenAddress, amount);
328 }
```

Please note that the related interface and event should be adjusted accordingly.

5.3. Modification of Vesting Information After Finalization

ID	IDX-003
Target	SuperDeedV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Very Low **Impact:** Low The configurator role can change the vesting information after the group state has been finalized to benefit more from the vesting. **Likelihood:** Low Besides the project owner, only the configurator role can change the vesting information and the project owner can change it back before starting the vesting.
Status	Resolved SuperLauncher team has resolved this issue as suggested by adding a validation on the `defineVesting()` function to prevent the vesting items of the group from being changed after being funded in commit `fd70fe63fefedf4bf8433fcd4b202a05bbf07044`.

5.3.1. Description

The `defineVesting()` function is used to set the vesting information in each group by the project owner or the configurator role.

SuperDeedV2.sol

```
53 function defineVesting(uint groupId, string memory groupName,
   DataType.VestingItem[] calldata vestItems) external  notLive
   onlyProjectOwnerOrConfigurator {
54     _check(groupId, groupName);
55     uint added = _groups().defineVesting(groupId, vestItems);
56     _recordHistory(DataType.ActionType.DefineVesting, added);
57 }
```

Once all details of the group are set, the project owner or the approver can set the group state to finalized by calling the `setGroupFinalized()` function.

SuperDeedV2.sol

```
70 function setGroupFinalized(uint groupId, string memory groupName) external
   notLive onlyProjectOwnerOrApprover {
71     _check(groupId, groupName);
72     _groups().setFinalized(groupId, groupName);
```

```
73      emit FinalizeGroup(msg.sender, groupId, groupName);
74      _recordHistory(DataType.ActionType.FinalizeGroup, groupId);
75  }
```

However, after the group state is finalized, the configurator can still change the vesting information before the vesting is started to gain more benefit from the vesting.

5.3.2. Remediation

Inspex suggest validating the group state in the `defineVesting()` function to validate that if group state is finalized, the vesting information cannot be changed, for example, as shown in line 54:

SuperDeedV2.sol

```
53  function defineVesting(uint groupId, string memory groupName,
    DataType.VestingItem[] calldata vestItems) external  notLive
    onlyProjectOwnerOrConfigurator {
54      _require(!group.state.finalized, "Can not change vesting information after
    the group is finalized");
55      _check(groupId, groupName);
56      uint added = _groups().defineVesting(groupId, vestItems);
57      _recordHistory(DataType.ActionType.DefineVesting, added);
58  }
```

5.4. Starting of Vesting Without Funded Group

ID	IDX-004
Target	SuperDeedV2
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SuperLauncher team has resolved this issue as suggested by adding the `_isFunded` state to check if at least one of the groups is funded, and adding a validation of the `_isFunded` state on the `startVesting()` function in commit `fd70fe63fefedf4bf8433fcd4b202a05bbf07044`.

5.4.1. Description

The `SuperDeedV2` contract has a design that every vesting group can be started simultaneously even when none of the groups are funded properly. The function for starting the vesting, `startVesting()`, only has a condition to check that the `tokenAddress` state is not the `ZERO_ADDRESS`. It doesn't check if the groups are finalized or funded.

SuperDeedV2.sol

```
149 function startVesting(uint startTime) external notLive
    onlyProjectOwnerOrApprover {
150
151     // Make sure that the asset address are set before start vesting
152     _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Set token address
    first");
153
154     if (startTime==0) {
155         startTime = block.timestamp;
156     }
157     _require(startTime >= block.timestamp, "Cannot back-date vesting");
158     _groups().vestingStartTime = startTime;
159     emit StartVesting(msg.sender, startTime);
160     _recordHistory(DataType.ActionType.StartVesting, startTime);
161 }
```

If none of the groups are funded when the vesting starts, the contract will not function properly, as the users will not be able to claim any token.

5.4.2. Remediation

Inspex suggests adding a validation to check if at least one group is funded and ready to be started. For example, a state variable, `_isFunded`, can be added into the `DataStore` contract.

DataStore.sol

```
27  uint internal _emergencyMaxAmount;
28  uint internal _emergencyExpiryTime;
29  bool internal _isFunded;
```

The `_isFunded` state should be set to `true` whenever a group is having the `group.state.funded` state set to `true`. There are two places that set the value of `group.state.funded`. They are at line 88 and line 120 on the SuperDeedV2 contract. So, we set the `_isFunded` state to `true` at line 89 on the `fundInForGroup()` function.

SuperDeedV2.sol

```
77  function fundInForGroup(uint groupId, string memory groupName, uint
    tokenAmount) external notLive onlyProjectOwnerOrApprover {
78      _check(groupId, groupName);
79      _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Invalid
    address");
80
81      // Check required token Amount is correct?
82      DataType.Group storage group = _groups().items[groupId];
83      _require(tokenAmount == group.info.totalEntitlement, "Wrong token amount");
84
85      // Group must be finalized and not yet fund in
86      _require(group.state.finalized, "Not yet finalized");
87      _require(!group.state.funded, "Already funded");
88      group.state.funded = true;
89      _isFunded = true; // set the new state to true
90
91      DataType.AssetType assetType = _asset().tokenType;
92      if (assetType == DataType.AssetType.ERC20) {
93          IERC20(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), tokenAmount);
94      } else if (assetType == DataType.AssetType.ERC1155) {
95          IERC1155(_asset().tokenAddress).safeTransferFrom(msg.sender,
    address(this), _asset().tokenId, tokenAmount, "");
96      } else {
97          // Verify that the amount has been deposied already ?
98          DataType.Erc721Handler storage handler = _store().erc721Handler;
99
```

```
100         uint totalDeposited721 = handler.erc721IdArray.length;
101         _require(totalDeposited721 >= (handler.numUsedByVerifiedGroups +
    tokenAmount), "Insufficient deposited erc721");
102         handler.numUsedByVerifiedGroups += tokenAmount;
103     }
104     emit FundInForGroup(msg.sender, groupId, groupName, tokenAmount);
105     _recordHistory(DataType.ActionType.FundInForGroup, groupId, tokenAmount);
106 }
```

Also, set the `_isFunded` state to `true` at the line 121 on the `fundInForGroupOverride()` function.

SuperDeedV2.sol

```
110 function fundInForGroupOverride(uint groupId, string memory groupName) external
    notLive onlyDaoMultiSig {
111
112     _check(groupId, groupName);
113     _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Invalid
    address");
114
115     // Check required token Amount is correct?
116     DataType.Group storage group = _groups().items[groupId];
117     // Group must be finalized and not yet fund in
118     _require(group.state.finalized, "Not yet finalized");
119     _require(!group.state.funded, "Already funded");
120     group.state.funded = true;
121     _isFunded = true; // set the new state to true
122
123     emit FundInForGroupOverrided(msg.sender, groupId, groupName);
124     _recordHistory(DataType.ActionType.FundInForGroupOverrided, groupId, 0);
125 }
```

On the `startVesting()` function, add the `_isFunded` state validation at line 153 on the `SuperDeedV2` contract to check if at least one of the groups is funded.

SuperDeedV2.sol

```
149 function startVesting(uint startTime) external notLive
    onlyProjectOwnerOrApprover {
150
151     // Make sure that the asset address are set before start vesting
152     _require(_asset().tokenAddress != Constant.ZERO_ADDRESS, "Set token address
    first");
153     _require(_isFunded, "At least one group needs to be funded");
154
155     if (startTime==0) {
156         startTime = block.timestamp;
157     }
```

```
158         _require(startTime >= block.timestamp, "Cannot back-date vesting");
159         _groups().vestingStartTime = startTime;
160         emit StartVesting(msg.sender, startTime);
161         _recordHistory(DataType.ActionType.StartVesting, startTime);
162     }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE