StationX Security Review

Pashov Audit Group

Conducted by: unforgiven, btk, juancito

June 5th 2024 - June 14th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **StationX-Network/smartcontract** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About StationX

StationX is a protocol that enables communities to create and manage on-chain memberships, pool capital, and run operations through secure multi-sig treasuries. Users can set up conditions for membership, handle day-to-day tasks, and raise funds easily within their Station.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - de6dbb69fd6ec51570899d6e110030d0d5ac602f

Scope

The following smart contracts were in scope of the audit:

- `Claims/`
- `LayerZero/`
- `Deployer`
- `emitter`
- `erc20dao`
- `erc721dao`
- `factory`
- `helper`
- `proxy`
- `zairdrop`
- `interfaces`

7. Executive Summary

Over the course of the security review, unforgiven, btk, juancito engaged with StationX to review StationX. In this period of time a total of **60** issues were uncovered.

Protocol Summary

Protocol Name	StationX
Repository	https://github.com/StationX-Network/smartcontract
Date	June 5th 2024 - June 14th 2024
Protocol Type	Community ownership protocol

Findings Count

Severity	**Amount**
Critical	3
High	12
Medium	15
Low	30
Total Findings	**60**

Summary of Findings

ID	Title	Severity	Status
[C-01]	The LayerZero implementation contract is the receiver of DAOs fees on cross-chain buy operations	Critical	Acknowledged
[C-02]	Native tokens in the DAOs are stuck	Critical	Acknowledged
[C-03]	Native tokens can be drained from the Factory contract	Critical	Acknowledged
[H-01]	Bypassing maxTokensPerUser limits for ERC721 DAO	High	Acknowledged
[H-02]	It is possible to override DAO details	High	Acknowledged
[H-03]	Users can break the lz communication	High	Acknowledged
[H-04]	Malicious dao owner can break lz communication	High	Acknowledged
[H-05]	Incorrect value will get sent to the commLayer	High	Acknowledged
[H-06]	Token-gated airdrops don't work with ERC20 DAOs	High	Acknowledged
[H-07]	Imprecise validation of deposit amounts	High	Acknowledged
[H-08]	Cross-chain deposits will fail when the deposit token is the native token	High	Acknowledged
[H-09]	DAO creators can avoid paying creation fees	High	Acknowledged
[H-10]	User address may not be accessible on other chains	High	Acknowledged
[H-11]	No refund nor retry mechanism for	High	Acknowledged

	failing cross-chain transactions		
[H-12]	Adversary can block the counterpart deployment of cross-chain DAOs	High	Acknowledged
[M-01]	Using hardcoded gas for cross-chain message	Medium	Acknowledged
[M-02]	Admin can't increase claim balance when hasAllowanceMechanism has been set in Claim contract	Medium	Acknowledged
[M-03]	mintGTToAddress() in ERC20DAO doesn't check distributionAmount limit	Medium	Acknowledged
[M-04]	Owner's share should be deducted from total payment in governance buy functions	Medium	Acknowledged
[M-05]	Cross-chain deployments failure	Medium	Acknowledged
[M-06]	Tokens with more than 18 decimals are not supported	Medium	Acknowledged
[M-07]	Cross-Chain and KYCed users will only pay a portion of the fees	Medium	Acknowledged
[M-08]	Unchecked Low-level Call when buying tokens	Medium	Acknowledged
[M-09]	Users can grief the owner by buying in chunks	Medium	Acknowledged
[M-10]	Initialization can be frontrun to set the owner	Medium	Acknowledged
[M-11]	LayerZero channel for deployments can be blocked due to reverts	Medium	Acknowledged
[M-12]	Non-blocking LayerZero cross-chain buy operations can be blocked	Medium	Acknowledged

[M-13]	Minting via mintGTToAddress may revert	Medium	Acknowledged
[M-14]	ERC721 max token per user limit can always be bypassed by one	Medium	Acknowledged
[M-15]	Adversary can emit any event that needs the EMITTER role without restrictions	Medium	Acknowledged
[L-01]	DAO contracts didn't use variable isGovernanceActive	Low	Acknowledged
[L-02]	updateProposalAndExecution() doesn't support sending ETH	Low	Acknowledged
[L-03]	Checking duplicate items in DAO chains list	Low	Acknowledged
[L-04]	Incorrect check in createERC20DAO and createERC721DAO	Low	Acknowledged
[L-05]	Malicious walletAddress can lock all user funds	Low	Acknowledged
[L-06]	Low-level call is not checked in claim()	Low	Acknowledged
[L-07]	Use call instead of transfer	Low	Acknowledged
[L-08]	Wrong data sent to Safe initializer	Low	Acknowledged
[L-09]	Users setting an incorrect tokenURI might end up losing their deposits	Low	Acknowledged
[L-10]	Unnecessary receivePayload call	Low	Acknowledged
[L-11]	Checking if the total NFT supplied is unlimited	Low	Acknowledged
[L-12]	Cross-chain deposit messages can be sent to the same chain	Low	Acknowledged

[L-13]	ERC721 DAO tokens minted in batch will always have the same tokenURI	Low	Acknowledged
[L-14]	Letting users set their tokenURI can be detrimental to the DAO	Low	Acknowledged
[L-15]	The DAO address is not validated when buying tokens	Low	Acknowledged
[L-16]	DAO creators can't set different tokens on destination chains	Low	Acknowledged
[L-17]	DAO creators can't deploy their DAO to other chains after the initial creation	Low	Acknowledged
[L-18]	- Storage variables changed with toggle functions should have public visibility	Low	Acknowledged
[L-19]	Missing events for important changes	Low	Acknowledged
[L-20]	Initializing totalClaimAmount with a value different than zero	Low	Acknowledged
[L-21]	Use safeTransfer for ERC20 transfers	Low	Acknowledged
[L-22]	Tokens may not be assigned a tokenURI on re-entrant calls	Low	Acknowledged
[L-23]	Rogue users can prevent minting tokens via mintGTToAddress	Low	Acknowledged
[L-24]	DAOs may not be able to receive NFTs	Low	Acknowledged
[L-25]	Any ERC1155 tokenId can be used for token-gated claims	Low	Acknowledged
[L-26]	Any Safe can emit changedSigners events on behalf of other DAOs	Low	Acknowledged
[L-27]	Hardcoded Safe fallback handler may	Low	Acknowledged

	not exist on some chains		
[L-28]	Missing view function to calculate the final value to provide	Low	Acknowledged
[L-29]	No way to change the owner or revoke roles	Low	Acknowledged
[L-30]	Inconsistent use of upgradeable contracts	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] The LayerZero implementation contract is the receiver of DAOs fees on cross-chain buy operations

Severity

Impact: High

Likelihood: High

Description

Whenever a cross-chain DAO is created via `createCrossChainERC20DAO()` or `createCrossChainERC721DAO()`, it assigns the `ownerAddress` as the `msg.sender`. The problem is that the `msg.sender` will always be the LayerZero Deployer contract:

```
function createCrossChainERC20DAO(...) {
@>      require(msg.sender == commLayer, "Caller not LZ Deployer");
        bytes memory _payload = abi.encode
          (_daoAddress, msg.sender, _numOfTokensToBuy, _tokenURI, _merkleProof);
        ccDetails[_daoAddress] =
            CrossChainDetails(
                _commLayerId,
                _depositChainIds,
                false,
@>              msg.sender,          // @audit ownerAddress
                _onlyAllowWhitelist
            );
    }
```

So, each time a cross-chain buy operation is performed, the fee will be attempted to be sent to the LayerZero Deployer, instead of the DAO owner address:

```
payable(
        ccDetails[_daoAddress].ownerAddress != address(0)
@>          ? ccDetails[_daoAddress].ownerAddress
            : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress
    ).call{value: ownerShare}("");
```

The fee won't be sent anywhere as the LayerZero Deployer doesn't have a `receive() payable {}` function, and the result from the `call` is not checked. In any case, the DAO owner's address will not receive the corresponding fees.

Recommendations

Allow the DAO creator to specify an `ownerAddress` for each cross-chain DAO deployment (as they may not have control over the same address on all chains).

[C-02] Native tokens in the DAOs are stuck

Severity

Impact: High. Native tokens from deposits are sent to the DAO, but can't be moved from there.

Likelihood: High. This will always happen for DAOs with native token deposits.

Description

Both `_buyGovernanceTokenERC20DAO()` and `_buyGovernanceTokenERC721DAO()` will send their native token deposits to the DAO address when configured so:

```
if (_daoDetails.depositTokenAddress == NATIVE_TOKEN_ADDRESS) {
        payable
          (_daoDetails.assetsStoredOnGnosis ? _daoDetails.gnosisAddress : _daoAddress)
            value: _totalAmount
        }("");
    }
```

The problem is that there is no function to withdraw, use, or transfer those native tokens.

Recommendations

Pass a `_value` parameter to the `updateProposalAndExecution()` function in both `ERC20DAO` and `ERC721DAO` to transfer the native tokens.

Also consider making the function `payable`, as it can be useful for the DAO to make external calls with `value`, but without using its deposits.

```
-   function updateProposalAndExecution(address _contract, bytes memory _data)
+   function updateProposalAndExecution
+ (address _contract, bytes memory _data, uint256 _value)
        external
        onlyGnosis(factoryAddress, address(this))
        nonReentrant
+       payable
    {
-       (bool success,) = _contract.call(_data);
+       (bool success,) = _contract.call{value: _value}(_data);
        require(success);
    }
```

[C-03] Native tokens can be drained from the Factory contract

Severity

Impact: High. Stolen assets.

Likelihood: High. Any user can do it.

Description

An adversary can perform an attack to steal all native tokens from the Factory contract by exploiting an error in the `crossChainBuy()` function.

Here's a simplified version of the error. It's present on the `value: msg.value - fees` calculation:

```
function crossChainBuy(...) {
        _buyGovernanceTokenERC20DAO(_daoAddress, _numOfTokensToBuy);

        fees = ((depositFees * platformFeeMultiplier) / 100);
@>      ICommLayer(_commLayer).sendMsg{value: msg.value - fees}
  (_commLayer, _payload, _extraParams);
    }
```

`sendMsg()` will send a LayerZero message with `value = msg.value - fees` and will refund the excess amount to an address provided by the user inside `_extraParams`.

Let's say that `msg.value = 10 ETH`, `fees = 0.1 ETH` and the LayerZero message requires another 0.1 ETH. So, LayerZero will receive 9.9 ETH and refund the amount not used. The user's final balance would be 9.8 ETH.

Note how the protocol fees stay in the Factory contract. This also happens for DAO deployments. Native token fees will remain in the contract until the protocol owner claims them. These are the tokens that can be stolen.

Notice how in `_buyGovernanceTokenERC20DAO()` (and the ERC721 function) the native tokens deposited are sent to the Safe/DAO, while some fee is sent to the DAO owner:

```
function _buyGovernanceTokenERC20DAO(...) {
        uint256 ownerShare =
          (_totalAmount * _daoDetails.ownerFeePerDepositPercent) / (FLOAT_HANDLER_TEN_

        if (_daoDetails.depositTokenAddress == NATIVE_TOKEN_ADDRESS) {
            checkDepositFeesSent(_daoAddress, _totalAmount + ownerShare);
@>          payable
  (_daoDetails.assetsStoredOnGnosis ? _daoDetails.gnosisAddress : _daoAddress).call{
                value: _totalAmount
            }("");

            payable(
                ccDetails[_daoAddress].ownerAddress != address(0)
                    ? ccDetails[_daoAddress].ownerAddress
                    : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress
@>          ).call{value: ownerShare}("");
        }
    }
```

These funds are sent from the Factory because it validates that the `msg.value` provided is enough to cover them all via `checkDepositFeesSent()`.

As an example, let's consider the deposited amount is 9 ETH, and the fees are 0.1 ETH. So 9.1 ETH will be transferred to the DAO/Safe/DAO owner. An adversary can even create a fake DAO to receive this.

Remember that as long as the contract has enough funds, it will also send the previously mentioned `value = msg.value - fees` to LayerZero, and finally refund the excess to the user (in our example 9.8 ETH).

So basically, the Factory contract is supplying the necessary native tokens for cross-chain deposits.

Recommendations

Calculate the remaining `value` correctly to send to LayerZero, considering if the deposit token is the native one, the deposit amount, the owner share, the deposit fees, and the KYC fees.

8.2. High Findings

[H-01] Bypassing `maxTokensPerUser` limits for ERC721 DAO

Severity

Impact: Medium

Likelihood: High

Description

When users call `buyGovernanceTokenERC721DAO()` code performs max cap checks and then calls `mintToken()`. The issue is that there's no reentrancy guard and when code wants to mint tokens one by one by using `_safeMint()` user can reenter the Factory contract again in ERC721's hook function and call `buyGovernanceTokenERC721DAO()` and buy more while bypassing the checks because `_tokenIdTracker` and `balanceOf(_to)` are still not updated fully. The impact is more severe in DAOs that have whitelists because one user can buy more than what he is supposed to.

```
function mintToken(
      address_to,
      stringcalldata_tokenURI,
      uint256_amount
    ) public onlyFactory(factoryAddress
        if (balanceOf(_to) + _amount > erc721DaoDetails.maxTokensPerUser) {
            revert MaxTokensMintedForUser(_to);
        }

        if (!erc721DaoDetails.isNftTotalSupplyUnlimited) {
            require(
                Factory(factoryAddress).getDAOdetails(address(
                  factoryAddress
                ).getDAOdetails(address
                  (this
        }
        for (uint256 i; i < _amount;) {
            _tokenIdTracker += 1;
            _safeMint(_to, _tokenIdTracker);
```

This is POC:

1. User1 would call `buyGovernanceTokenERC721DAO()` to buy 10 tokens while max token for each user is 15.
2. When code wants to mint the first token for user and calls `_safeMint()` and User1's address would be called by hook function.
3. User1 would call `buyGovernanceTokenERC721DAO()` again to buy 10 more tokens and this times because the previous buy amounts has not been added to the `balanceOf(_to)` so the checks would pass.
4. As result User1 would buy 20 tokens while the max limit for each user was 15.

Recommendations

Use reentrancy guard or use check-effect-interact pattern.

[H-02] It is possible to override DAO details

Severity

Impact: High

Likelihood: Medium

Description

Code uses `create` to deploy DAO token in Deployer contract, so the DAO address only depends on Deployer contract addresses and its deployment nonce. If two chains have the same Deployer contract address then DAO created in those chains would have the same addresses. This would cause congestion when some DAOs deploy on both of those chains. An attacker can use this to override the DAO's details in the other chain. So when a DAO uses multi-chain deployments it would be possible to deploy the same DAO in the cross-chain and the result would be that the real DAO's information in the factory can be overwritten.

This is the POC:

1. User1 wants to deploy his DAO in chain1 and chain2. He creates a tx deploys his DAO in chain1 and uses chain2 as the second chain. The DAO address will be determined only by the result of `create` opcode which is based on Deployer contract address and its deployment `nonce`.

2. If Chain1 and Chain2 had the same Deployer contract address then the attacker can create a DAO in chain2 whose addresses would be the same and the attacker would set chain1 as a side chain for his DAO.
3. As a result the Factory contract in chian2 would send a message to Chain1 to deploy the attacker-specified DAO in the chain1 in the same DAO address of the User1 and it would cause a rewrite of the User1's DAO.

Recommendations

Make sure DAOs have different addresses in different chains.

[H-03] Users can break the lz communication

Severity

Impact: High

Likelihood: Medium

Description

The LayerZeroImpl contract uses the NonBlockingLzApp from the LZ SDK to store all failed messages for future retries:

```
function _blockingLzReceive(
      uint16_srcChainId,
      bytesmemory_srcAddress,
      uint64_nonce,
      bytesmemory_payload
    )
        internal
    {
        (bool success, bytes memory reason) = address(this).excessivelySafeCall(
            gasleft(),
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );

        if (!success) {
            _storeFailedMessage
              (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }
```

The function uses up to `gasleft()` gas and reads up to 150 bytes of returndata using `excessivelySafeCall()`. Due to the 63/64 rule, only 1/64 of the remaining gas is left for storing the failed message if `nonblockingLzReceive()` uses all its allocated gas. LayerZeroImpl forwards 600k gas to the endpoint:

```
endpoint.send{value: msg.value}(
            _dstChainId,
            abi.encodePacked(dstCommLayer[uint16(_dstChainId)], address(this)),
            _payload,
            payable(refundAd),
            address(0x0),
            abi.encodePacked(uint16(1), uint256(600000)) // 600k
        );
```

If all 63/64 gas is used, only ~9000 gas remains for storing the failed message, which is insufficient since a single zero to non-zero SSTORE costs 22.1k gas. This results in a revert in the Endpoint try/catch handling.

```
try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}
          (_srcChainId, _srcAddress, _nonce, _payload) {
            // success, do nothing, end of the message delivery
        } catch (bytes memory reason) {
            // revert nonce if any uncaught errors/exceptions if the ua chooses
            // the blocking mode
            storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64
              (_payload.length), _dstAddress, keccak256(_payload));
            emit PayloadStored
              (_srcChainId, _srcAddress, _dstAddress, _nonce, _payload, reason);
        }
```

That is the portion that stores the payload and blocks the channel. Since gas is capped to gasLimit here, there's no risk of not leaving enough gas to store the failure in `storedPayload`, the Relayer is just expected to provide a small extra buffer for running the logic before and after `lzReceive()`.

A malicious user can exploit this by wasting all the allocated gas using the `onERC721Received()`.

- Bob is a smart contract with a malicious code in his `onERC721Received()` that will waste all the allocated gas.
- Bob calls `crossChainBuy()` to buy DAO tokens, triggering the following call stack:
  - `commLayer.sendMsg()`
    - `endpoint.send()`
      - `commLayer.lzReceive()`
- `lzReceive()` calls `_nonblockingLzReceive()`, which calls `factory.crossChainMint()`.
- `crossChainMint()` will make some sanity checks and then call:
  - `dao.mintToken()`
    - `_safeMint()`
      - `onERC721Received()`
- Bob `onERC721Received()` hook will waste the entire allocated gas, causing `_blockingLzReceive()` to fail and attempts to store the failed message with insufficient gas.
- The revert bubbles up to the Endpoint try/catch handling, resulting in a blocked pathway.

Recommendations

A simple and effective solution is to use mint instead of safeMint. Make sure to document this clearly:

> If `to` is a smart contract, it must implement {IERC721Receiver-onERC721Received} hook.

[H-04] Malicious dao owner can break lz communication

Severity

Impact: High

Likelihood: Medium

Description

The LayerZeroImpl contract uses the NonBlockingLzApp from the LZ SDK to store all failed messages for future retries:

```
function _blockingLzReceive(
      uint16_srcChainId,
      bytesmemory_srcAddress,
      uint64_nonce,
      bytesmemory_payload
    )
        internal
    {
        (bool success, bytes memory reason) = address(this).excessivelySafeCall(
            gasleft(),
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );

        if (!success) {
            _storeFailedMessage
              (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }
```

The function uses up to `gasleft()` gas and reads up to 150 bytes of returndata using `excessivelySafeCall()`. Due to the 63/64 rule, only 1/64 of the remaining gas is left for storing the failed message if `nonblockingLzReceive()` uses all its allocated gas. LayerZeroImpl forwards 600k gas to the endpoint:

```
endpoint.send{value: msg.value}(
            _dstChainId,
            abi.encodePacked(dstCommLayer[uint16(_dstChainId)], address(this)),
            _payload,
            payable(refundAd),
            address(0x0),
            abi.encodePacked(uint16(1), uint256(600000)) // 600k
        );
```

If all 63/64 gas is used, only ~9000 gas remains for storing the failed message, which is insufficient since a single zero to non-zero SSTORE costs 22.1k gas. This results in a revert in the Endpoint try/catch handling.

```
try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}
          (_srcChainId, _srcAddress, _nonce, _payload) {
            // success, do nothing, end of the message delivery
        } catch (bytes memory reason) {
            // revert nonce if any uncaught errors/exceptions if the ua chooses
            // the blocking mode
            storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64
              (_payload.length), _dstAddress, keccak256(_payload));
            emit PayloadStored
              (_srcChainId, _srcAddress, _dstAddress, _nonce, _payload, reason);
        }
```

That is the portion that stores the payload and blocks the channel. Since gas is capped to gasLimit here, there's no risk of not leaving enough gas to store the failure in `storedPayload`, the Relayer is just expected to provide a small extra buffer for running the logic before and after `lzReceive()`.

A malicious DAO owner can exploit this by setting up multiple tokens to gate the community using a large array of tokens, wasting the allocated gas.

- Bob, an ERC20 DAO owner, sets up an excessively large list of tokens for token gating.
- Bob calls `crossChainBuy()` to buy DAO tokens, triggering the following call stack:
  - `commLayer.sendMsg()`
    - `endpoint.send()`
      - `commLayer.lzReceive()`
- `lzReceive()` calls `_nonblockingLzReceive()`, which calls `factory.crossChainMint()`.
- `crossChainMint()` loops through all tokens, causing an "Out Of Gas" error due to the large array.
- `_blockingLzReceive()` fails and attempts to store the failed message with insufficient gas.
- The revert bubbles up to the Endpoint try/catch handling, resulting in a blocked pathway.

Recommendations

Limit the number of tokens that can be added by the owner, for example, to 10:

```
function setupTokenGating(
        address[] calldata _tokens,
        Operator _operator,
        uint256[] calldata _value,
        address payable _daoAddress
    ) external payable onlyAdmins(daoDetails[_daoAddress].gnosisAddress) {
        require(_value.length == _tokens.length, "Length mismatch");
        require(_tokens.length <= 10, "Large array");

        tokenGatingDetails[_daoAddress] = TokenGatingCondition
          (_tokens, _operator, _value);

        daoDetails[_daoAddress].isTokenGatingApplied = true;
    }
```

[H-05] Incorrect value will get sent to the commLayer

Severity

Impact: High

Likelihood: High

Description

The `factory.createERC20DAO()` function creates cross-chain DAOs using LayerZero. Users can specify msg.value, which is distributed across multiple chains. For example, if there are 5 chains and msg.value is 1000, each chain receives 200. This value is forwarded to the commLayer.

The _depositChainIds array is intended to manage the distribution across chains. For instance, if a user wants to create a DAO on the primary chain and 2 secondary chains:

- Bob wants to create a DAO on the primary chain (Ethereum) and 2 secondary chains (Binance and Arbitrum One).
- Bob wants to forward 250 value to each secondary chain, so he must pass 500.
- The depositChainIds array would be:

```
uint256[] memory depositChainIds = new uint256[](3);
        depositChainIds[0] = 1; // Ethereum
        depositChainIds[1] = 56; // Binance
        depositChainIds[2] = 42161; // Arbitrum One
```

However, there are two issues:

1. **Incorrect Division:** The value calculation divides by 3 instead of 2, incorrectly including the primary chain.
2. **Decreasing Value:** The value shrinks with each iteration, while _depositChainIds.length remains constant. For example:

- First Iteration:

```
i = 0 (i < _depositChainIds.length - 1)
value = msg.value / _depositChainIds.length = 500 / 3 = 166
```

- Second Iteration:

```
i = 1 (i < _depositChainIds.length - 1)
value = msg.value / _depositChainIds.length = 334 / 3 = 111
```

Only 277 (166 + 111) is forwarded instead of the intended 500. The remaining 223 is left in the factory, withdrawable only by the owner.

Recommendations

Update the value calculation in createERC20DAO() and createERC721DAO() functions:

```
ICommLayer(commLayer).sendMsg{value: (msg.value - fees) /
                  (_depositChainIds.length - (i + 1))}(
                    commLayer, _payload, abi.encode
                      (_depositChainIds[i + 1], msg.sender)
                );
```

- First Iteration:

```
i = 0 (i < _depositChainIds.length - 1)
value = msg.value / (_depositChainIds.length - (i + 1)) = 500 / 2 = 250
```

- Second Iteration:

```
i = 1 (i < _depositChainIds.length - 1)
value = msg.value / (_depositChainIds.length - (i + 1)) = 250 / 1 = 250
```

[H-06] Token-gated airdrops don't work with ERC20 DAOs

Severity

Impact: Medium

Likelihood: High

Description

The function to `claim()` tokens in the `Claim` contract first checks that the `daoToken` supports an interface via `supportsInterface()`:

```
if (claimSettingsMemory.permission == CLAIM_PERMISSION.TokenGated) {
@>      if (IERC20Extended(claimSettingsMemory.daoToken).supportsInterface
  (0xd9b67a26)) {
            if (
                IERC20Extended(claimSettingsMemory.daoToken).balanceOf
                  (msg.sender, _tokenId)
                    < claimSettingsMemory.tokenGatingValue
            ) revert InsufficientBalance();
        } else {
            if (IERC20(claimSettingsMemory.daoToken).balanceOf
              (msg.sender) < claimSettingsMemory.tokenGatingValue) {
                revert InsufficientBalance();
            }
        }
```

The problem is that the `ERC20DAO` doesn't implement the `supportsInterface()` function, so it will revert every time it is called for an ERC20 DAO token.

Note that this is not an issue for `ERC721DAO` as it inherits from `ERC721Upgradeable`, which inherits from `ERC165Upgradeable`, which implements `supportsInterface()()`.

Recommendations

It would be enough to make `ERC20DAO` inherit from `ERC165Upgradeable if the intention is to allow only DAO tokens`.

If any other ERC20 tokens are expected to be used to check the balance for an airdrop, it would be recommended to check interface support via `ERC165Checker::supportsERC165()` which won't revert when used.

[H-07] Imprecise validation of deposit amounts

Severity

Impact: Medium

Likelihood: High

Description

All the minting operations `buyGovernanceTokenERC20DAO()`, `buyGovernanceTokenERC721DAO()`, and `crossChainMint()` have a check to validate that the deposited tokens are below a certain limit representing the total amount raised:

```
// For ERC20 DAOs
if (daoBalance + _totalAmount >
  (_daoDetails.pricePerToken * _daoDetails.distributionAmount) / 1e18) {
    revert AmountInvalid("daoBalance", daoBalance + _totalAmount);
}

For ERC721 DAOs
checkAmountValidity(daoBalance, _totalAmount, _daoDetails.pricePerToken,
_daoDetails.distributionAmount);

function checkAmountValidity(uint256 _daoBalance, uint256 _totalAmount,
uint256 _pricePerToken, uint256 _distributionAmount
  ) internal pure {
    if (_distributionAmount != 0) {
        uint256 _maxAllowedAmount = _pricePerToken * _distributionAmount;
        if (_daoBalance + _totalAmount > _maxAllowedAmount) {
            revert AmountInvalid("daoBalance", _daoBalance + _totalAmount);
        }
    }
}
```

Note how all of these checks are dependent on `daoBalance`, which signals the balance at a specific moment the DAO contract has:

```
uint256 daoBalance;
    if (daoDetails[_daoAddress].depositTokenAddress == NATIVE_TOKEN_ADDRESS) {
        daoBalance = _daoAddress.balance;
    } else {
        daoBalance = IERC20
          (daoDetails[_daoAddress].depositTokenAddress).balanceOf(_daoAddress);
    }
```

This is problematic, as the balance of the tokens in the DAO contract doesn't reflect the total deposited tokens over time needed to calculate the total amount raised. This value can go up if the DAO receives tokens, or go down if the tokens are moved out.

Furthermore, depending on configuration, the deposited tokens might never go to the DAO, but to the Safe wallet. In such a case the DAO balance could be zero, and no limit would be applied ever.

Recommendations

Keep track of the total deposited tokens in a storage variable instead of tracking the temporary balance of the DAO contract, and perform validations with it.

[H-08] Cross-chain deposits will fail when the deposit token is the native token

Severity

Impact: High

Likelihood: Medium

Description

The `crossChainMint()` function calculates the `_daoBalance` as:

```
uint256 _daoBalance = IERC20(_daoDetails.depositTokenAddress).balanceOf
  (_daoAddress);
```

The problem is that the cross-chain DAO can be deployed with the native token (defined in the contract as address: `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`).

So, anytime a user tries to buy cross-chain DAO tokens, they will deposit tokens but the transaction on the destination chain will always revert as it will try to perform a `balanceOf()` call on a non-existing contract.

Recommendations

Check the native token balance in case the cross-chain DAO was deployed with it.

[H-09] DAO creators can avoid paying creation fees

Severity

Impact: Medium

Likelihood: High

Description

Both `createERC20DAO()` and `createERC721DAO()` verify that enough native tokens were sent as fees via `checkCreateFeesSent()`:

```
function checkCreateFeesSent
  (uint256[] calldata _depositChainIds) internal returns (uint256) {
    uint256 fees = createFees * _depositChainIds.length;
    require(msg.value >= fees, "Insufficient fees");
    return fees;
}
```

The problem is that when `_depositChainIds.length == 0` no creation fees are paid. This is possible because there are no checks on the mentioned functions to prevent this.

Furthermore, there is an explicit `_depositChainIds.length != 0` condition that prevents any out-of-bounds error that would make the transaction revert as well:

```
if (_depositChainIds.length != 0) {
    for (uint256 i; i < _depositChainIds.length - 1;) {
        // ...
        ICommLayer(commLayer).sendMsg{value:
          (msg.value - fees) / _depositChainIds.length}(
            commLayer, _payload, abi.encode(_depositChainIds[i + 1], msg.sender)
        );
    // ...
}
```

Recommendations

Validate that creation fees are paid when a DAO is only deployed to the source chain, with a check like `require(_depositChainIds.length > 0)` for example.

[H-10] User address may not be accessible on other chains

Severity

Impact: High

Likelihood: Medium

Description

A smart wallet is deployed in one chain and has a specific address. But it is possible that trying to deploy a smart wallet with the same address won't be possible on other chains. For example, if the deployer is not available on the destination chain, or if it is not possible to set the same salt/noce. Also on chains like zkSync that `create` operations will always lead to different addresses with the same deployment parameters. In any of these cases, the user won't be able to control the same address on both chains.

Some cross-chain functions assume that the same address in the source chain will also be controlled on the destination chain.

`crossChainBuy()` sets the receiver of the DAO tokens to `msg.sender`. The tokens might be sent to an address they don't control.

`createERC20DAO()` and `createERC721DAO()` set the same `admins` as the source chain for their cross-chain Safe. So admins might not be able to control the Safe, and not be able to transfer locked funds in the DAO (among other admin functions).

Recommendations

- For `crossChainBuy()`: A simple mitigation would be allowing cross-chain buy operations just for EOAs. A more robust but complex mitigation would let users set a specific receiver on the destination chain. However, it would have to consider any side effects of such refactoring, as it would open the possibility of minting tokens on behalf of other users, which isn't possible now.
- For `createERC20DAO()`: Allow DAO creators to provide specific `admins` for each deployed cross-chain DAO

[H-11] No refund nor retry mechanism for failing cross-chain transactions

Severity

Impact: High

Likelihood: Medium

Description

When performing cross-chain transactions, the transaction may always revert to the destination chain under certain circumstances or for certain values.

Users will deposit tokens or pay fees in the source chain that will be lost when those reverts occur on the destination chain and can't be recovered by them.

For the cross-chain buy operation: The user will deposit tokens on the source chain via `crossChainBuy()`. If `crossChainMint()` reverts, no DAO token will be minted. This may happen for different reasons:

- If the amount is not valid (depends on DAO balance, token price, and distribution amount in the DAO chain)
- If `maxTokensPerUser` would be surpassed
- If `distributionAmount` would be surpassed
- If the total deposit for the user is below or above a limit
- If the user is not whitelisted
- If token gating is applied
- User errors

It is important to note that most of these conditions depend on moving values, and any off-chain validation might be true when sending the source chain transaction, but may not hold when the cross-chain transaction is executed. Validations on the source chain would not be sufficient, as the source of truth is ultimately on the chain to which the DAO contract was deployed.

In addition, this could theoretically happen for cross-chain DAO deployments too, but not likely. A possible revert might be on an invalid `_depositTokenAddress` that can't be queried for its `decimals()` (although the deployment wouldn't make sense with an invalid address). Other checks are performed in `_createERC20DAO()` or `_createERC721DAO()`, but with the same values as for the source chain, so they would have reverted on the original transaction. Other statements in the destination chain should not revert but bear in mind that they would lead to the scenario described in this report.

Recommendations

Provide a refund mechanism for users who made deposits but didn't receive their DAO tokens. Also, consider allowing to retry messages that may temporarily revert.

[H-12] Adversary can block the counterpart deployment of cross-chain DAOs

Severity

Impact: Medium

Likelihood: High

Description

The `deploySAFE()` function in `Deployer` attempts to create a Safe for the DAO via `SafeProxyFactory::createProxyWithNonce()`. It uses a `try/catch` pattern, but it will always revert if another Safe with the same `nonce` was created beforehand on that chain.

Notice how a `revert` inside a `catch` block will make the whole transaction revert:

```
uint256 nonce = getNonce(_daoAddress);
do {
    try ISafe(safe).createProxyWithNonce
      (singleton, _initializer, nonce) returns (address _deployedSafe) {
        SAFE = _deployedSafe;
    } catch Error(string memory reason) {
        nonce = getNonce(_daoAddress);
        revert SafeProxyCreationFailed(reason);
    } catch {
        revert SafeProxyCreationFailed("Safe proxy creation failed");
    }
} while (SAFE == address(0));
```

An adversary can directly call SafeProxyFactory::createProxyWithNonce() on the destination chain while the cross-chain transaction is being processed by LayerZero.

Using the same `nonce` generated for the victim `_daoAddress()` and the same `initialize` parameters will guarantee it will create a Safe with the same address.

This is problematic since it will make the cross-chain transaction revert when trying to execute `deploySAFE()` in `createCrossChainERC20DAO()`:

```
function createCrossChainERC20DAO(...) {
    address _safe = IDeployer(deployer).deploySAFE
      (_admins, _safeThreshold, _daoAddress);

    _createERC20DAO(...);
    ccDetails[_daoAddress] = CrossChainDetails
      (_commLayerId, _depositChainIds, false, msg.sender, _onlyAllowWhitelist);
}
```

It will be impossible to finish the creation of the DAO counterpart on the destination chain, and buy operations will not be available there.

Note: There's also a secondary, less severe impact of this finding that involves frontrunning the transaction on the source chain, and preventing the creation of any DAO altogether. In this case, there is no risk as no DAO was created on any chain, but it can be considered a griefing attack. The recommendation covers this scenario as well.

Recommendations

One possible solution is to remove the `try/catch` block, precompute the expected address of the Safe proxy, check if it exists (has code), and only create it if not.

If the contract for the corresponding address was already deployed, it shouldn't pose any risks, as it had to be created for the corresponding `daoAddress`, with the expected admins, and other parameters set on its initializer.

8.3. Medium Findings

[M-01] Using hardcoded gas for cross-chain message

Severity

Impact: Medium

Likelihood: Medium

Description

Contracts LayerZeroImpl and LayerZeroDeployer uses hardcoded gas amounts when they send cross-chain messages through LayerZero bridges. The issue is that different destination chains may require different amount of gas and also the message itself can cause different amount of gas. When users create DAO they specify multiple lists and the length of those lists can have impact in the cross-chain message gas fee. As result the cross-chain DAO deployment would be lost for some chains and some DAOs.

Recommendations

Have some mechanisms to estimate the require gas amount that considers `baseGas`, `intrinsicGas` and `executionGas`.

[M-02] Admin can't increase claim balance when `hasAllowanceMechanism` has been set in Claim contract

Severity

Impact: Low

Likelihood: High

Description

When users creates DAO they specify the `claimBalance` and later they can increase it. The issue is that when `hasAllowanceMechanism` is set there's no way to increase `claimBalance` because `depositTokens()` won't allow it. This would cause constant `claimBalance` that admin set during the DAO creation and it won't be possible to increase airdrop total amount.

Recommendations

Allow admins to increase `claimBalance` when `hasAllowanceMechanism` is set.

[M-03] `mintGTToAddress()` in ERC20DAO doesn't check `distributionAmount` limit

Severity

Impact: Medium

Likelihood: Medium

Description

While `mintGTToAddress()` in ERC721 and Factory contract check for `distributionAmount` limit the ERC20DAO doesn't check for `distributionAmount` limit and it would be possible for admins to mint unlimited amount of tokens.

Recommendations

Check for `distributionAmount` in `mintGTToAddress()` in ERC20DAO.

[M-04] Owner's share should be deducted from total payment in governance buy functions

Severity

Impact: Low

Likelihood: High

Description

When users wants to buy governance token code calculates the total amount and then calculates the owner's share. The issue is that code transfers `totalAmount + ownerShare` from user. So as result users pays more than real price of the tokens because code doesn't subtract owner's share from total amount.

Recommendations

Code should calculate `totalAmount` and only transfer that amount from users. The `ownerShare` should be part of total amount and not an extra amount.

[M-05] Cross-chain deployments failure

Severity

Impact: Medium

Likelihood: Medium

Description

When code wants to create cross-chain DAO deployment message it divides the total ETH amount equally between cross-chain messages. The issue is that different destination chains may require different cross-chain fee. Code should allow user to specify bridge fee amount for each cross-chain message. In the current design user have to pay based on maximum bridge fee amount.

Recommendations

Allow user to specify cross-chain bridge fee amounts for each chain.

[M-06] Tokens with more than 18 decimals are not supported

Severity

Impact: Medium

Likelihood: Medium

Description

The current design of the `amountToSD()` and `amountToLD()` functions limits compatibility to tokens with 18 or fewer decimals. If a DAO is created using a token with more than 18 decimals as the `depositTokenAddress`, the DAO creation will fail due to an underflow issue during the `convertRate` calculation. While this poses no financial risk, it restricts the factory's adaptability within the broader DeFi ecosystem, preventing its use with such tokens.

For instance, tokens like YAMv2, which has 24 decimals, would cause the computation to attempt 18 - 24, resulting in an underflow and unsuccessful DAO creation.

Recommendations

Consider checking if decimals > 18 and normalize the value by div the decimals difference. Here is an example implementation:

> https://github.com/code-423n4/2022-06-connext-findings/issues/204#issuecomment-1170453579

[M-07] Cross-Chain and KYCed users will only pay a portion of the fees

Severity

Impact: Medium

Likelihood: Medium

Description

The factory contract calculates fees for cross-chain DAOs and KYC users as follows:

```
function checkDepositFeesSent
      (address _daoAddress, uint256 _totalAmount) internal {
        if
          (ccDetails[_daoAddress].depositChainIds.length > 1 || isKycEnabled[_daoAddre
            require(msg.value >= _totalAmount + ((
              msg.value>=_totalAmount+
            ) / 100
        } else {
            require
              (msg.value >= _totalAmount + depositFees, "Insufficient fees");
        }
    }
```

For example, if `depositFees` is 100 and the `platformFeeMultiplier` is 125, the factory will charge 125 as fees when the DAO is either cross-chain or KYCed. The sponsor indicated that KYC is a separate functionality, and the factory should charge additional fees when KYC is enabled. However, this is not the case currently. If the DAO is both cross-chain and KYCed, the factory only applies the `platformFeeMultiplier` once instead of twice.

Recommendations

Consider updating the function as follows:

```
function checkDepositFeesSent
      (address _daoAddress, uint256 _totalAmount) internal {
        uint256 fees = (depositFees * platformFeeMultiplier) / 100;
        if
          (ccDetails[_daoAddress].depositChainIds.length > 1 && isKycEnabled[_daoAddre
            require(msg.value >= _totalAmount +
              (fees * 2)), "Insufficient fees";
        } else if
          (ccDetails[_daoAddress].depositChainIds.length > 1 || isKycEnabled[_daoAddre
            require(msg.value >= _totalAmount + fees, "Insufficient fees");
        } else {
            require
              (msg.value >= _totalAmount + depositFees, "Insufficient fees");
        }
    }
```

[M-08] Unchecked Low-level Call when buying tokens

Severity

Impact: Low

Likelihood: High

Description

In the `_buyGovernanceTokenERC20DAO()` and `_buyGovernanceTokenERC721DAO()` functions, the success of low-level calls to both `gnosisAddress` and `ownerAddress` is not verified. This oversight means that when these calls fail, the failure will go unnoticed, leaving the funds in the factory without being distributed to `gnosisAddress` or `ownerAddress` .This results in neither party receiving their share of the payment.

Recommendations

Check the return value of the low-level calls within both `_buyGovernanceTokenERC20DAO()` and `_buyGovernanceTokenERC721DAO()` functions. If the call fails, the transaction should revert. Additionally, ensure that the gas limit for the call to `ownerAddress` is set appropriately to prevent a gas griefing attack.

[M-09] Users can grief the owner by buying in chunks

Severity

Impact: Low

Likelihood: High

Description

The factory contract supports all ERC20 tokens, allowing DAO creators to add any token as a payment method for their DAO. When users buy tokens using the `buyGovernanceTokenERC20DAO()` function, the owner takes a portion of the payment as a fee:

```
uint256 _totalAmount =
          (_numOfTokensToBuy * _daoDetails.pricePerToken) / 1e18;

        if (_totalAmount == 0) {
            revert AmountInvalid("_numOfTokensToBuy", _totalAmount);
        }

        uint256 ownerShare =
          (_totalAmount * _daoDetails.ownerFeePerDepositPercent) / (FLOAT_HANDLER_TEN_
```

However, users can exploit this calculation by making small, fragmented purchases, causing the `ownerShare` to round down to zero. This issue is particularly problematic for tokens with low decimal precision, such as GUSD, which has 2 decimals. Consider the following scenario:

- DAO Settings:
  - `Dao.depositTokenAddress = GUSD`
  - `Dao.ownerFeePerDepositPercent = 100`
  - `pricePerToken = 10e2`
- Exploit Execution:
  - Bob, a malicious user aware of the bug, aims to exploit the system.
  - Bob calls `buyGovernanceTokenERC20DAO()` with `numOfTokensToBuy = 9.9e16`.
  - Calculation:
    - `_totalAmount = (9.9e16 * 10e2) / 1e18 = 99`
    - `ownerShare = (99 * 100) / 10000 = 0`
  - The ownerShare rounds down to zero, meaning the owner receives no fees for each purchase of 9.9e16 tokens.
  - The trick is that `_totalAmount * _daoDetails.ownerFeePerDepositPercent` must be less than `FLOAT_HANDLER_TEN_4` (i.e. 10_000).

Recommendations

To prevent this exploit, consider updating the check as follows:

```
if (_totalAmount == 0 ||
          (_totalAmount * _daoDetails.ownerFeePerDepositPercent) < FLOAT_HANDLER_TEN_4
            revert AmountInvalid("_numOfTokensToBuy", _totalAmount);
        }
```

[M-10] Initialization can be frontrun to set the owner

Severity

Impact: Medium

Likelihood: Medium

Description

Contracts utilizing an initialization function are vulnerable to frontrunning attacks during deployment. An attacker could intercept the deployment process and assign themselves as the contract owner, as ownership is granted to the msg.sender. While StationX could redeploy the compromised contracts, this would incur financial costs. Additionally, if StationX deploys contracts without realizing they have been pre-initialized by an attacker, users could unknowingly engage with a compromised system from the outset.

Affected contracts are:

- Deployer
- Emitter
- Factory
- ClaimEmitter
- ClaimFactory
- LayerZeroDeployer
- LayerZeroImpl

Recommendations

To mitigate this risk, set the owner in the constructor and restrict the initialize() function to the owner only for the listed contracts.

[M-11] LayerZero channel for deployments can be blocked due to reverts

Severity

Impact: Medium

Likelihood: Medium

Description

The `LayerZeroDeployer` contract uses LayerZero v1, which has a blocking nature by default, and its receiver function doesn't implement a non-blocking

pattern. Any revert during the cross-chain execution will block the whole channel, preventing future DAO deployments, until the protocol admin unlocks it via `forceResume()`.

The cross-chain calls `createCrossChainERC20DAO()` and `createCrossChainERC721DAO()` should not revert in theory, but an adversary can craft a transaction that will pass validations and succeed on the source chain, but not on the destination chain.

One way to do so is by passing a large array of `_admins` to the cross-chain transaction. At some point, the function will revert with an "Out of Gas" error given the loop size:

```
address _safe = IDeployer(deployer).deploySAFE
  (_admins, _safeThreshold, _daoAddress);

for (uint256 i; i < _admins.length;) {
    IEmitter(emitterAddress).newUserCC(

                _daoAddress, _admins[i], _depositTokenAddress, 0, block.timestamp, 0,
    );

    unchecked {
        ++i;
    }
}
```

In the source chain, the adversary can send any amount of gas they want, so the transaction can succeed even with a reasonably big number of `_admins`. The problem is in the destination chain as the gas is limited to `2_000_000`:

```
endpoint.send{value: msg.value}(
        uint16(_dstChainId),
        abi.encodePacked(dstCommLayer[uint16(_dstChainId)], address(this)),
        _payload,
        payable(_refundAd),
        address(0x0),
@>      abi.encodePacked(uint16(1), uint256(2_000_000))
    );
```

So, it is possible to make a transaction run successfully on the source chain and revert to the destination chain, blocking all subsequent cross-chain transactions given LayerZero v1 default behavior.

Recommendations

Validate that the `_admins` array is below the max limit on the source chain when calling `createERC20DAO()` and `createERC721DAO()`. Also, consider

applying a non-blocking pattern to the cross-chain deployments.

[M-12] Non-blocking LayerZero cross-chain buy operations can be blocked

Severity

Impact: Medium

Likelihood: Medium

Description

When a cross-chain buy operation is performed, a LayerZero message is sent with a fixed `600_000` gas limit to execute in the destination chain:

```
endpoint.send{value: msg.value}(
        _dstChainId,
        abi.encodePacked(dstCommLayer[uint16(_dstChainId)], address(this)),
        _payload,
        payable(refundAd),
        address(0x0),
@>      abi.encodePacked(uint16(1), uint256(600_000))
    );
```

LayerZero v1 has a blocking behavior by default, and `LayerZeroImpl` attempts to implement the non-blocking pattern via code:

```
function _blockingLzReceive(
      uint16_srcChainId,
      bytesmemory_srcAddress,
      uint64_nonce,
      bytesmemory_payload
    ) internal {
@>      (bool success, bytes memory reason) = address(this).excessivelySafeCall(
            gasleft(),
            150,
            abi.encodeWithSelector(
              this.nonblockingLzReceive.selector,
              _srcChainId,
              _srcAddress,
              _nonce,
              _payload
            )
        );

        if (!success) {
@>          _storeFailedMessage
  (_srcChainId, _srcAddress, _nonce, _payload, reason);
        }
    }

    function _storeFailedMessage(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload,
        bytes memory _reason
    ) internal virtual {
@>      failedMessages[_srcChainId][_srcAddress][_nonce] = keccak256(_payload);
@>      emit MessageFailed(_srcChainId, _srcAddress, _nonce, _payload, _reason);
    }
```

In theory, this should work as expected by trying to execute the cross-chain transaction and saving any failed message for later processing in case of failure.

But if for some reason the transaction reverts on the `failedMessages[_srcChainId][_srcAddress][_nonce] = keccak256(_payload);` statement, LayerZero will consider the whole cross-chain transaction as failed, and it will block the channel for future messages.

Setting a value from zero to a non-zero value costs at least 20,000 gas. So if the remaining gas up to that point is less than that, the transaction will revert, and LayerZero will block the channel.

Buy messages are executed with a fixed 600,000 gas limit, as previously mentioned. According to EIP-150 63/64 or ~590k would be forwarded to the external call. If all gas is used in `excessivelySafeCall()`, that would leave ~9,000 gas for the rest of the execution. This isn't enough to cover storing `failedMessages` + emitting the `MessageFailed` event and the transaction will revert.

`excessivelySafeCall()` calls `crossChainMint()`, and it should not be possible to make that function spend 590k gas with normal usage, but an adversary can provide for example a big enough `_merkleProof` array that would consume the gas when hashing over and over in a big loop. They can also send a big enough `_tokenURI` that would be too expensive to store when minting the token.

Also note that even if not spending all the gas inside `excessivelySafeCall()`, storing a big payload + emitting a message with a big payload will cost much more than 20,000 gas, and can also make the transaction revert.

Recommendations

Validate that the payload for the messages is below a reasonable size for the use case and the gas limit provided. Also consider validating max lengths of arrays, bytes, and strings before sending cross-chain transactions.

Note that this attack can also affect cross-chain deployments. It is recommended to apply the same recommendations there as well.

[M-13] Minting via mintGTToAddress may revert

Severity

Impact: Medium

Likelihood: Medium

Description

When minting ERC721 DAO tokens via `mintGTToAddress()` a check is performed to validate that the user won't end up with more than the max limit:

```
uint256 length = _userAddress.length;
    for (uint256 i; i < length;) {
        for (uint256 j; j < _amountArray[i];) {
@>          if (balanceOf
  (_userAddress[i]) + _amountArray[i] > erc721DaoDetails.maxTokensPerUser) {
@>              revert MaxTokensMintedForUser(_userAddress[i]);
@>          }

            _tokenIdTracker += 1;
            _safeMint(_userAddress[i], _tokenIdTracker);
            _setTokenURI(_tokenIdTracker, _tokenURI[i]);
            unchecked {
                ++j;
            }
        }
    }
```

So, for example, if the user has 0 NFTs, the max limit is 10, and 10 tokens are minted it should succeed as `0 (balance) + 10 (amount) <= 10 (maxTokensPerUser)`.

The problem is that this check will be performed after each mint. So, on the next iteration the balance will be one, and `1 (balance) + 10 (amount) <= 10 (maxTokensPerUser)` will be false. This will make the transaction revert, and the tokens won't be minted.

Recommendations

Consider moving the `if` check outside of the `j` loop:

```
uint256 length = _userAddress.length;
    for (uint256 i; i < length;) {
+       if (balanceOf
+ (_userAddress[i]) + _amountArray[i] > erc721DaoDetails.maxTokensPerUser) {
+           revert MaxTokensMintedForUser(_userAddress[i]);
+       }
        for (uint256 j; j < _amountArray[i];) {
-           if (balanceOf
- (_userAddress[i]) + _amountArray[i] > erc721DaoDetails.maxTokensPerUser) {
-               revert MaxTokensMintedForUser(_userAddress[i]);
-           }

            _tokenIdTracker += 1;
            _safeMint(_userAddress[i], _tokenIdTracker);
            _setTokenURI(_tokenIdTracker, _tokenURI[i]);
            unchecked {
                ++j;
            }
        }
    }
```

[M-14] ERC721 max token per user limit can always be bypassed by one

Severity

Impact: Low

Likelihood: High

Description

Both `transferFrom()` and `safeTransferFrom()` functions in `ERC721DAO` have an off-by-one error when calculating the max amount of tokens a user can have:

```
require(balanceOf(to) <= erc721DaoDetails.maxTokensPerUser);
```

This is also inconsistent with the limit validation performed in `mintToken()`, where it is applied correctly.

This leads to users being able to have more tokens than they should. For example, a DAO that would like to limit their tokens to one per user would allow them to have two instead.

Recommendations

Consider making this change to both `transferFrom()` and `safeTransferFrom()`:

```
-   require(balanceOf(to) <= erc721DaoDetails.maxTokensPerUser);
+   require(balanceOf(to) < erc721DaoDetails.maxTokensPerUser);
```

[M-15] Adversary can emit any event that needs the EMITTER role without restrictions

Severity

Impact: Low

Likelihood: High

Description

The `Emitter` contract has two functions that grant an `EMITTER` role to the DAO proxies. This role is granted to every DAO created via `createERC20DAO()` or `createERC721DAO()` in the factory contract.

```
function createDaoErc20(..., address _proxy, ...) {
        _grantRole(EMITTER, _proxy);
    }

    function createDaoErc721(..., address _proxy, ...) {
        _grantRole(EMITTER, _proxy);
    }
```

The problem is that the DAOs have a function that allows their Safe to execute a call to any contract with any data:

```
function updateProposalAndExecution(
      address_contract,
      bytesmemory_data
    ) external onlyGnosis(factoryAddress, address(this
        (bool success,) = _contract.call(_data);
        require(success);
    }
```

This can be leveraged by an adversary to make calls to the `Emitter` contract to emit deceiving events that may exploit some off-chain logic. Here's an example of the 26 affected functions:

```
function deposited(address _daoAddress, address _depositor,
        address _depositTokenAddress, uint256 _amount, ...
    ) external onlyRole(EMITTER) {
        emit Deposited(_daoAddress, _depositor, _depositTokenAddress, _amount,
            _timestamp, _ownerFee, _adminShare);
    }
```

Recommendations

A simple approach would be to disable calls to the `Emitter` contract in `updateProposalAndExecution()` for both `ERC20DAO` and `ERC721DAO`.

```
function updateProposalAndExecution(
      address_contract,
      bytesmemory_data
    ) external onlyGnosis(factoryAddress, address(this
+       require(_contract != emitterContractAddress);
        (bool success,) = _contract.call(_data);
        require(success);
    }
```

8.4. Low Findings

[L-01] DAO contracts didn't use variable `isGovernanceActive`

Variable `isGovernanceActive` didn't get used in the code. Code shouldn't allow for governance operations when `isGovernanceActive` is set to `False`.

[L-02] `updateProposalAndExecution()` doesn't support sending ETH

Function `updateProposalAndExecution()` is supposed to execute DAO operations but the issue is that it doesn't support transferring ETH and as result some functionalities won't be able to be executed in that DAO.

[L-03] Checking duplicate items in DAO chains list

There's no check that DAO chain deployment list have duplicate items and if it has duplicate items then the DAO may have unexpected behavior in the side chains because the configs for that DAO would be set two times in the side chains and those configs may not be the same. Code would create two cross-chain message for the same destination chain and those message would receive asynchronously and cause configuration change for DAO. For example DAO's deposit tokens may change in the side chain.

[L-04] Incorrect check in createERC20DAO and createERC721DAO

In the `createERC20DAO()` and `createERC721DAO()` functions, a loop iterates through the user's deposit chain IDs only if the length is not zero:

```
if (_depositChainIds.length != 0) {
            for (uint256 i; i < _depositChainIds.length - 1;) {
```

However, this approach is flawed. The loop initiates even when `_depositChainIds.length == 1`, which signifies no cross-chain creation. This leads to an emission of the `createCCDao()` event when no cross-chain creation occurs. Consider updating the check as follows:

```
if (_depositChainIds.length > 1) {
            for (uint256 i; i < _depositChainIds.length - 1;) {
```

[L-05] Malicious walletAddress can lock all user funds

The claim contract includes a `hasAllowanceMechanism` boolean, enabling the claim creator to allow users to claim directly from a specified wallet. However, if this `walletAddress` becomes compromised and revokes the allowance, user funds will be locked. Additionally, the claim creator currently cannot update this address. To address this issue, consider adding a function that allows updating the `walletAddress`.

[L-06] Low-level call is not checked in claim()

In the claim() function, the success of the low-level call is not being verified. If the call fails, the factory will not receive the claimFee, and the fee will remain locked in the claim contract:

```
if (claimAmount[msg.sender] == 0) {
            if (msg.value != IFactory(factory).claimFee()) {
                revert InvalidAmount();
            }
            payable(factory).call{value: msg.value}("");
        }
```

Consider checking the return value as shown below:

```
(bool success, ) = payable(factory).call{value: msg.value}("");
            require(success, "Call failed");
```

[L-07] Use call instead of transfer

solidity `transfer()` function is used in `disburseNative()` for native ETH transfer. The transfer and send functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The recipient smart contract does not implement a payable function.
- The recipient smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The recipient smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.
- Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Use call instead of transfer and add a gas limit for each call.

[L-08] Wrong data sent to Safe initializer

The Safe initializer is declared with a `"0x"` string literal. Note that this is not the same as setting it as an empty `bytes` value.

This value uses the `data` to set up a module. This doesn't pose any risks with the current codebase as no modules are set, but it would be advisable to fix it to prevent any problems if the code is updated to include a module.

```
bytes memory _initializer = abi.encodeWithSignature(
        "setup
          (address[],uint256,address,bytes,address,address,uint256,address)",
        _admins,
        _safeThreshold,
        0x0000000000000000000000000000000000000000,
@>      "0x",
        0xf48f2B2d2a534e402487b3ee7C18c33Aec0Fe5e4,
        0x0000000000000000000000000000000000000000,
        0,
        0x0000000000000000000000000000000000000000
    );
```

Consider changing `"0x"` to `""`.

[L-09] Users setting an incorrect tokenURI might end up losing their deposits

When buying tokens via `crossChainBuy()` if a user sets an empty `tokenURI` for an ERC721 DAO, or sets a non-empty `tokenURI` for an ERC20 DAO, the transaction will succeed on the source chain, but it will revert on the destination chain because it would try to mint tokens for a wrong type of DAO.

Consider checking that `tokenURI` is valid for the type of DAO the user is trying to buy tokens on `crossChainBuy()` so that the transaction reverts without any deposits.

[L-10] Unnecessary receivePayload call

The `lzReceive()` functions in `LayerZeroDeployer` and `LayerZeroImpl` have an unnecessary statement used for testing purposes by Layer Zero.

```
if (keccak256(abi.encodePacked((_payload))) == keccak256(abi.encodePacked(
  (bytes10("ff"))))) {
    endpoint.receivePayload(1, bytes(""), address(0x0), 1, 1, bytes(""));
}
```

Consider removing it from both contracts.

[L-11] Checking if the total NFT supplied is unlimited

The function currently checks `_distributionAmount != 0`, but the corresponding attribute that signals unlimited supplied is `isNftTotalSupplyUnlimited`.

Consider checking for `isNftTotalSupplyUnlimited == true` instead of `_distributionAmount != 0`. This would prevent any user error.

[L-12] Cross-chain deposit messages can be sent to the same chain

The `crossChainBuy()` function doesn't validate that the cross-chain deposit isn't sent to the same chain. This might be used to attempt to avoid cross-chain deposit fee multipliers since `ccDetails[_daoAddress]` is empty for the source chain:

```
if
  (ccDetails[_daoAddress].depositChainIds.length > 1 || isKycEnabled[_daoAddress]) {
    fees = ((depositFees * platformFeeMultiplier) / 100);crossChainMint?
}
```

Consider checking that the message is not sent to the same chain, or always charging the cross-chain multiplier for `crossChainBuy()`.

[L-13] ERC721 DAO tokens minted in batch will always have the same tokenURI

Both `mintToken()` and `mintGTToAddress()` from `ERC721DAO` allow batch minting tokens for one user. The problem is that they will all have the same `tokenURI`. The `tokenURI` differentiates one NFT from another, and it is often important to do so. It is also worth mentioning that this attribute can't be changed by the user later.

Consider passing different token URIs for each batch-minted token.

[L-14] Letting users set their tokenURI can be detrimental to the DAO

The `tokenURI` of an NFT usually contains the unique attributes of a token and represents its non-fungibility.

By letting users set any arbitrary `tokenURI`, provide deceiving information, set invalid values that may not be rendered correctly on UIs, or users can copy the values from other tokens. Their pricing will also be affected as they won't have any unique attributes to differentiate them from one another.

Consider letting DAO admins set individual `tokenURI` values instead of users, or let them choose any of these options.

[L-15] The DAO address is not validated when buying tokens

The `_daoAddress` parameter is not validated in any of the functions: `buyGovernanceTokenERC20DAO()`, `buyGovernanceTokenERC721DAO()`, or `crossChainBuy()`. This doesn't pose any risks on the current codebase because of unexpected reverts deeper into the functions.

Consider validating that `_daoAddress` is a DAO deployed by the factory to prevent any possible issues on future code changes.

[L-16] DAO creators can't set different tokens on destination chains

The DAO creation functions `createERC20DAO()` and `createERC721DAO()` accommodate values related to token deposit amounts or pricing to 18 decimals:

```
if (_depositChainIds.length != 0) {
        for (uint256 i; i < _depositChainIds.length - 1;) {
            bytes memory _payload = abi.encode(
                _commLayerId,
                _distributionAmount,
@>              amountToSD(_depositTokenAddress[0], _pricePerToken),
@>              amountToSD(_depositTokenAddress[0], _minDepositPerUser),
@>              amountToSD(_depositTokenAddress[0], _maxDepositPerUser),
                // ...
@>              _depositTokenAddress[i + 1],
```

And then on the destination they accommodate them back to the decimals corresponding to the deposit token there via `createCrossChainERC20DAO()` and `createCrossChainERC721DAO()`:

```
_createERC20DAO(
        _distributionAmount,
@>      amountToLD(_depositTokenAddress, _pricePerToken),
@>      amountToLD(_depositTokenAddress, _minDepositPerUser),
@>      amountToLD(_depositTokenAddress, _maxDepositPerUser),
        // ...
@>      _depositTokenAddress,
```

The problem is that the code assumes that DAO creators will use the "same" token on all chains, which limits their operations (by "same" it means the counterpart on the destination chain, like Ethereum USDT <> Polygon USDT).

Most importantly, some tokens don't exist in all chains, so DAO creators will have to use other ones (with different pricing and min/max deposit limits).

It would be recommended to allow DAO creators to set different deposit tokens on each chain, with their respective price, and deposit limit values.

[L-17] DAO creators can't deploy their DAO to other chains after the initial creation

DAOs can be created via `createERC20DAO()`, and `createERC721DAO()`. This will also trigger the creation of the DAO counterpart in other chains via LayerZero. The problem is that after the initial creation, there is no possible way to deploy the DAO on new chains.

It would be suggested to add a function to allow DAO admins to deploy the DAO on other chains after the initial creation.

[L-18] - Storage variables changed with toggle functions should have public visibility

The storage variables `isPaused`, and `isKycEnabled` have `private` visibility.

```
bool private isPaused;
mapping(address => bool) private isKycEnabled;
```

The problem is that there is no easy way to check their current states off-chain or by other contracts on-chain. This is also problematic as their state is changed via toggle functions:

```
function togglePaused() external {
    _onlyOwners();
    isPaused = !isPaused;
}

function toggleKYC(address _daoAddress) external payable onlyAdmins
  (daoDetails[_daoAddress].gnosisAddress) {
    isKycEnabled[_daoAddress] = !isKycEnabled[_daoAddress];
}
```

It would be recommended to make those storage variables `public`, or define setter functions instead of toggle ones.

[L-19] Missing events for important changes

Some functions make important changes that will be difficult to track by off-chain tools because they don't emit any events.

Consider adding an event to:

- `Claim`: `claimAllPending()`
- `ClaimFactory`: `setEmitter()`, `changeClaimPrice()`, `changeClaimFee()`, `changeClaimImplementation()`, `withdrawFunds()`
- `LayerZeroDeployer`: `changeFactory()`, `changeEndpoint()`, `setDestination()`
- `LayerZeroImpl`: `changeFactory()`, `changeEndpoint()`, `setDestination()`
- `Deployer`: `defineContracts()`
- `ERC20DAO`: `toggleOnlyAllowWhitelist()`, `updateGovernanceActive()`
- `ERC721DAO`: `toggleOnlyAllowWhitelist()`, `updateGovernanceActive()`
- `Factory`: `changeOwner()`, `togglePaused()`, `defineTokenContracts()`, `setupTokenGating()`, `disableTokenGating()`, `updateFees()`, `toggleKYC()`

[L-20] Initializing `totalClaimAmount` with a value different than zero

The `totalClaimAmount` value in the `Claim` contract is supposed to keep internal accounting information. However, it may be set to a non-zero value when on initialization as there is no check to prevent the admin from doing so.

This will lead to improper accounting and can lead to reverts or undesired effects in the functions using it.

Consider checking `_claimSettings.claimAmountDetails.totalClaimAmount == 0` when initializing the contract.

[L-21] Use safeTransfer for ERC20 transfers

Some tokens may not be compatible with the `IERC20::transfer()` / `IERC20::transferFrom()` interfaces and the transaction would revert.

The `airDropToken()` function in `zairdrop` uses `transferFrom()`:

```
IERC20(_airdropTokenAddress).transferFrom
  (msg.sender, _members[i], _airdropAmountArray[i]);
```

The `rescueFunds()` function in `factory` uses `transfer()`:

```
IERC20(tokenAddr).transfer(_owner, balance);
```

Consider using `safeTransferFrom` in these cases.

[L-22] Tokens may not be assigned a tokenURI on re-entrant calls

The `mintToken()` function in `ERC721DAO` doesn't respect the Checks-Effects-Interactions pattern. If a re-entrant call is made, and more tokens are minted `_tokenIdTracker += 1` will be incremented first for all of the newly minted

tokens, and after that `_setTokenURI()` will be called with the last `_tokenIdTracker` value. This means that it will set the `tokenURI` N times for the last token, and it won't set it for the previous ones:

```
for (uint256 i; i < _amount;) {
@>      _tokenIdTracker += 1;
@>      _safeMint(_to, _tokenIdTracker);
@>      _setTokenURI(_tokenIdTracker, _tokenURI);
        unchecked {
            ++i;
        }
    }
```

Consider making storage changes before interactions in `mintToken()`, and also `mintGTToAddress()`:

```
for (uint256 i; i < _amount;) {
        _tokenIdTracker += 1;
+       _setTokenURI(_tokenIdTracker, _tokenURI);
        _safeMint(_to, _tokenIdTracker);
-       _setTokenURI(_tokenIdTracker, _tokenURI);
        unchecked {
            ++i;
        }
    }
```

[L-23] Rogue users can prevent minting tokens via mintGTToAddress

The `mintGTToAddress()` function mints tokens for many users via `_safeMint()`. This function makes an `onERC721Received()` callback for each user. If any of these users make the transaction revert, or if attempting to transfer a token to a contract not implementing an ERC721 receiver, the whole transaction will revert, and no tokens will be minted for any user.

The minting could still be retried without the conflicting user. But if the attack is desired to be prevented altogether, it would be suggested to implement a pull-over-push pattern that lets each user claim their token.

[L-24] DAOs may not be able to receive NFTs

Neither `ERC20DAO` nor `ERC721DAO` implement the receivers for ERC721 or ERC1155 tokens. This will prevent the DAOs from receiving NFTs sent via `safeTransferFrom()` to them, as they expect the corresponding `onERC721Received()`, `onERC1155Received()`, and `onERC1155BatchReceived()` functions to be implemented.

It is suggested that both DAO contracts implement those receivers or inherit them from ERC721Holder and `ERC1155Holder`.

[L-25] Any ERC1155 tokenId can be used for token-gated claims

For `ERC1155` NFTs, different IDs usually have unique attributes that represent them. The `claim()` function in the `Claim` contract allows to pass any `_tokenId` value for the token-gated validation:

```
function claim(..., uint256 _tokenId) external payable nonReentrant {
        if (IERC20Extended(claimSettingsMemory.daoToken).supportsInterface
          (0xd9b67a26)) {
            if (
@>              IERC20Extended(claimSettingsMemory.daoToken).balanceOf
  (msg.sender, _tokenId)
                    < claimSettingsMemory.tokenGatingValue
            ) revert InsufficientBalance();
        }
    }
```

Consider the admins of the `Claim` contracts to allow setting the specific `tokenId` values valid for token gating, and validate them in the `claim()` function.

[L-26] Any Safe can emit changedSigners events on behalf of other DAOs

The `emitSignerChanged()` function in `ERC20DAO` and `ERC721DAO` allows the Safe wallet to emit events on behalf of any other DAO, just by passing a different `_dao` value. This can be used by an adversary to mess up the indexed values off-chain.

```
function emitSignerChanged(
      address_dao,
      address_signer,
      bool_isAdded
    ) external onlyGnosis(factoryAddress, address(this
        Emitter(emitterContractAddress).changedSigners(
@>          _dao,
            _signer,
            _isAdded);
    }
```

Consider replacing `_dao` with `address(this)` on the `changedSigners()` call.

[L-27] Hardcoded Safe fallback handler may not exist on some chains

The Safe initializer sets the fallback handler as the hardcoded `0xf48f2B2d2a534e402487b3ee7C18c33Aec0Fe5e4` address, which corresponds to the `CompatibilityFallbackHandler`.

Note that this address is not guaranteed to be the same in all chains as seen in the Safe Docs. One example can be zkSync, where that address is instead `0x2f870a80647BbC554F3a0EBD093f11B4d2a7492A`.

Consider adding a setter function to specify the correct value on each chain.

[L-28] Missing view function to calculate the final value to provide

Buying tokens, or deploying DAOs incurs many variable fees, depending on KYC, cross-chain transactions, and fee percentages.

No view function allows users to easily request how much it will cost them to pay accordingly.

This can be quite challenging for a UI without an easy way to access this data. Consider adding such a function.

[L-29] No way to change the owner or revoke roles

Some contracts have a sole privileged user with owner/admin capabilities. It would be recommended to add a function to transfer the ownership to another address, both for administrative reasons and for risk prevention.

This issue is present in: `LayerZeroImpl`, `LayerZeroDeployer`, `Deployer`, `zairdrop`.

In the case of the `emitter` contract, it is assigning a custom `ADMIN` role. It would be recommended that instead, it assigns the `DEFAULT_ADMIN_ROLE` role from OpenZeppelin, which allows granting and revoking other roles as well.

On a similar line, `erc20dao`, and `erc721` can grant a `RefundModule` role, but don't have a way of removing it. Consider adding a function to do so.

[L-30] Inconsistent use of upgradeable contracts

Some upgradeable contracts don't use the upgradeable versions of the OpenZeppelin contracts as recommended. These contracts may contain initializers to work properly or reserve gaps in storage for future upgrades.

Consider using the upgradeable OpenZeppelin contract version for:

- `erc20dao`: `AccessControlUpgradeable`, `ReentrancyGuardUpgradeable`
- `erc721dao`: `AccessControlUpgradeable`, `ERC2981Upgradeable`
- `emitter`: `AccessControlUpgradeable`
- `ClaimFactory`: `AccessControlUpgradeable`
- `ClaimEmitter`: `AccessControlUpgradeable`
- `Claim`: `AccessControlUpgradeable`, `ReentrancyGuardUpgradeable`

Note how the current commit doesn't build because of a "missing" override for `_contextSuffixLength()` in `erc20dao` and `erc721dao`. This override is not necessary with the suggested change of contracts.

Also, note that the `_msgSender()` and `_msgData()` overrides in `erc20dao` and `erc721dao` would not be needed anymore (as long as no additional functionality is added) because the contract's conflict was solved. These can be safely removed.

Also consider disabling initializers on the upgradeable implementations as recommended for the following contracts: `Claim`, `ClaimEmitter`, `ClaimFactory`, `LayerZeroDeployer`, `LayerZeroImpl`, `factory`, `Deployer`, and `emitter`.

Finally check that the OpenZeppelin initializers are called on the corresponding `initialize()` function of each contract:

- `Claim`: `__AccessControl_init()`, `__ReentrancyGuard_init()`
- `ClaimEmitter`: `__AccessControl_init()`
- `ClaimFactory`: `__AccessControl_init()`
- `emitter`: `__AccessControl_init()`
- `erc20dao`: `__AccessControl_init()`, `__ReentrancyGuard_init()`
- `erc721dao`: `__AccessControl_init()`, `__ERC2981_init()`

An uninitialized implementation contract can be initialized by an attacker, it’s recommended to invoke the _disableInitializers() function in the constructor to prevent the implementation contract from being used by the attacker.