SALUS SECURITY

APR 2024

CODE SECURITY ASSESSMENT

STATIONX

Overview

Project Summary

- Name: StationX
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/StationX-Network/smartcontract
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	StationX
Version	v3
Type	Solidity
Dates	Apr 22 2024
Logs	Apr 17 2024; Apr 21 2024; Apr 22 2024

Vulnerability Summary

Total High-Severity issues	5
Total Medium-Severity issues	3
Total Low-Severity issues	4
Total informational issues	1
Total	13

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Lack of access control on createCrossChainERC721DAO/ERC20Dao() leads to malicious manipulation of dao details	High	Access Control	Resolved
2	Lack of access control on sendmsg() function allows malicious payload to be sent	High	Access Control	Resolved
3	Unchecked return values of low level calls	High	Data Validation	Resolved
4	Anyone can unblock the queue of messages	High	Access Control	Resolved
5	ZetaMessage can be maliciously constructed by anyone	High	Access Control	Resolved
6	Users can get dao token for free	Medium	Business Logic	Resolved
7	Anyone can take zeta tokens out from the ZetaImpl contract	Medium	Access Control	Resolved
8	Factory unexpected losses createFee	Medium	Business Logic	Resolved
9	Possible dos when creating dao	Low	Denial of Service	Resolved
10	Use call instead of transfer for native tokens transfer	Low	Business logic	Resolved
11	Key events can be triggered arbitrarily	Low	Logging	Resolved
12	Incorrect boundary checking	Low	Data Validation	Resolved
13	Implementation contract could be initialized by everyone	Informational	Configuration	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Lack of access control on createCrossChainERC721DAO/ERC20Dao() leads to malicious manipulation of dao details

Severity: High	Category: Access Control

Target:
- contracts/factory.sol

Description

contracts/factory.sol:L544-L606

```
function createCrossChainERC721DAO(
    uint16 _commLayerId,
    uint256 _ownerFeePerDepositPercent,
    uint256 _depositTime,
    uint256 _quorumPercent,
    uint256 _thresholdPercent,
    uint256 _safeThreshold,
    uint256[] memory _depoitChainIds,
    address _daoAddress,
    address _depositTokenAddress,
    address[] memory _admins,
    uint256 _maxTokensPerUser,
    uint256 _distributionAmount,
    uint256 _pricePerToken,
    bool _onlyAllowWhitelist,
    bytes32 _merkleRoot
) external {
    address _safe = IDeployer(deployer).deploySAFE(
        _admins,
        _safeThreshold,
        _daoAddress
    );

    _createERC721DAO(
        _ownerFeePerDepositPercent,
        _depositTime,
        _quorumPercent,
        _thresholdPercent,
        _daoAddress,
        _depositTokenAddress,
        _safe,
        _maxTokensPerUser,
        _distributionAmount,
        amountToLD(_depositTokenAddress, _pricePerToken),
        true,
        _merkleRoot
    );
    ...
}
```

Anyone can create a dao with the specified dao address and parameters.

contracts/factory.sol:L609-L660

```
function _createERC721DAO(
    uint256 _ownerFeePerDepositPercent,
    uint256 _depositTime,
    uint256 _quorumPercent,
    uint256 _thresholdPercent,
    address _daoAddress,
    address _depositTokenAddress,
    address _gnosisAddress,
    uint256 _maxTokensPerUser,
    uint256 _distributionAmount,
    uint256 _pricePerToken,
    bool _assetsStoredOnGnosis,
    bytes32 _merkleRoot
) private {
    ...
    daoDetails[_daoAddress] = DAODetails(
        _pricePerToken,
        _distributionAmount,
        0,
        0,
        _ownerFeePerDepositPercent,
        _depositTime,
        _depositTokenAddress,
        _gnosisAddress,
        _merkleRoot,
        true,
        false,
        _assetsStoredOnGnosis
    );
}
```

This means that a malicious user can use an already created dao address as a parameter and overwrite it with the code above.

The result is that the configuration stored in the daoDetails will be free to be modified by a malicious user.

Recommendation

Consider designing appropriate access control for this function. For example restricting calls that must come from cross-chain bridges:

```
require(msg.sender == commLayer, "Caller not LZ Deployer");
```

Status

The team has resolved this issue in commit 01afe79b.

2. Lack of access control on sendmsg() function allows malicious payload to be sent

Severity: High	Category: Access Control

Target:
- contracts/LayerZero/LayerZeroImpl.sol

Description

contracts/LayerZero/LayerZeroImpl.sol:L91-L105

```
function sendMsg(address _destination, bytes calldata _payload, bytes memory
extraParams) public payable {
    (uint16 _dstChainId, address refundAd) = abi.decode(extraParams, (uint16, address));

    uint64 nextNonce = endpoint.getOutboundNonce(_dstChainId, address(this)) + 1;

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

The sendMsg() function is used to send a message to the endpoint.

Lack of access control can lead to users being able to fake messages and send them to the destination chain, thus destroying the authenticity of cross-chain messages.

Recommendation

Consider designing appropriate access control for this function.

Status

The team has resolved this issue in commit 4f66ae98.

3. Unchecked return values of low level calls

Severity: High	Category: Data Validation

Target:
- contracts/LayerZero/LayerZeroImpl.sol

Description

contracts/LayerZero/LayerZeroImpl.sol:L122-L130

```
function _blockingLzReceive(uint16 _srcChainId, bytes memory _srcAddress, uint64 _nonce,
bytes memory _payload)
    internal
{
    (bool success, bytes memory reason) = address(this).excessivelySafeCall(
        gasleft(),
        150,
        abi.encodeWithSelector(this.nonblockingLzReceive.selector, _srcChainId,
_srcAddress, _nonce, _payload)
    );
}
```

The cross-chain execution process may revert due to various checks, but the return value of the low-level call is not checked in _blockingLzReceive.

This means that when a cross-chain message fails to execute in the target chain, _blockingLzReceive() still executes successfully.

Once the protocol mistakenly believes that the message executed successfully, the user's funds may be permanently locked up on the original chain.

Recommendation

It is recommended to check the return values of low-level calls and handle exceptions correctly. For example, you can modify the code based on the LayerZero sample code.

Status

The team has resolved this issue in commit 2b0ab218.

4. Anyone can unblock the queue of messages

Severity: High	Category: Access Control
Target: - contracts/LayerZero/LayerZeroImpl.sol	

Description

contracts/LayerZero/LayerZeroImpl.sol:L167-L169

```
function forceResume(uint16 _srcChainId, bytes memory _from) external {
    endpoint.forceResumeReceive(_srcChainId, _from);
}
```

The message queue in LayerZero can be unlocked by using the forceResumeReceive() function.
However, in the forceResume() function, there is no access control, resulting in anyone being able to call the function in order to cancle cross-chain messages.

This allows a malicious user to cancel any cross-chain message.

Recommendation

Consider adding access control to only allow owner/multisig to unblock the queue of messages if something unexpected happens.

Status

The team has resolved this issue in commit 4f66ae98.

5. ZetaMessage can be maliciously constructed by anyone

Severity: High	Category: Access Control
Target: - contracts/ZetaChain/ZetaDeployer.sol	

Description

contracts/ZetaChain/ZetaDeployer.sol:L75-L160

```
function onZetaMessage(
    ZetaInterfaces.ZetaMessage calldata zetaMessage
) external override {
    ...
    IFactory(factory).createCrossChainERC721DAO(
        _commLayerId,
        _ownerFeePerDepositPercent,
        _depositTime,
        _quorumPercent,
        _thresholdPercent,
        _safeThreshold,
        _depoitChainIds,
        _daoAddress,
        _depositTokenAddress,
        _admins,
        _maxTokensPerUser,
        _distributionAmount,
        _pricePerToken,
        _onlyAllowWhitelist,
        _merkleRoot
    );
    ...
}
```

The onZetaMessage() is used to accept messages from the zeta endpoint. However, the function lacks access control, resulting in anyone being able to construct a message and call the function.
And the protocol will mistakenly believe that the message is real and valid and create the corresponding dao.

Recommendation

Consider designing appropriate access control for this function.

Status

The team has resolved this issue in commit d9f2ffe.

6. Users can get dao token for free

Severity: Medium	Category: Business logic

Target:
- contracts/factory.sol

Description

In the StationX protocol, a user can purchase a daoToken with a depositToken. However, due to a loss of precision issue, it may result in a user being able to acquire a daoToken without paying for a depositToken.

contracts/factory.sol:L772-L831

```
function _buyGovernanceTokenERC20DAO(
    address payable _daoAddress,
    uint256 _numOfTokensToBuy
) private {
    uint256 _totalAmount = (_numOfTokensToBuy * _daoDetails.pricePerToken) / 1e18;

    uint256 ownerShare = (_totalAmount *
        _daoDetails.ownerFeePerDepositPercent) / (FLOAT_HANDLER_TEN_4);

    if (_daoDetails.depositTokenAddress == NATIVE_TOKEN_ADDRESS) {
        checkDepositFeesSent(_daoAddress, _totalAmount + ownerShare);
        payable(
            _daoDetails.assetsStoredOnGnosis
                ? _daoDetails.gnosisAddress
                : _daoAddress
        ).transfer(_totalAmount);
        payable(
            ccDetails[_daoAddress].ownerAddress != address(0)
                ? ccDetails[_daoAddress].ownerAddress
                : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress
        ).transfer(ownerShare);
    } else {
        checkDepositFeesSent(_daoAddress, 0);
        IERC20(_daoDetails.depositTokenAddress).safeTransferFrom(
            msg.sender,
            _daoDetails.assetsStoredOnGnosis
                ? _daoDetails.gnosisAddress
                : _daoAddress,
            _totalAmount
        );
        IERC20(_daoDetails.depositTokenAddress).safeTransferFrom(
            msg.sender,
            ccDetails[_daoAddress].ownerAddress != address(0)
                ? ccDetails[_daoAddress].ownerAddress
                : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress,
            ownerShare
        );
    }
    ...
}
```

The above highlighted code is used to calculate the number of depositTokens that should be paid for the number of DAOs _numOfTokensToBuy.
However, due to a potential loss of precision, this may result in _totalAmount being calculated as 0, but the contract will still mint the _numOfTokensToBuy amount of dao

tokens for the user.

For example, when pricePerToken is 1e16, if the user buys 99wei of daoToken, the user will not need to pay depositToken to the contract.

Recommendation

It is recommended to revert the transaction when the _totalAmount calculation results is 0. And take other actions to protect the loss of precision.

Status

The team has resolved this issue in commit 2b0ab218.

7. Anyone can take zeta tokens out from the ZetaImpl contract

Severity: Medium	Category: Access Control
Target: - contracts/ZetaChain/ZetaImpl.sol	

Description

contracts/ZetaChain/ZetaImpl.sol:L114-L116

```
function transferZeta(uint amount) external {
    _zetaToken.transfer(msg.sender, amount);
}
```

The zetaToken is used to pay cross-chain fees on Zeta, and a external function is provided in ZetaImpl.sol to extract the zetaToken.
This means that anyone can take out the zetaToken that the team has pre-existing in the contract.

Recommendation

Consider designing appropriate access control for this function.

Status

The team has resolved this issue in commit 2b0ab218.

8. Factory unexpected losses createFee

Severity: Medium	Category: Business logic
Target: - contracts/factory.sol	

Description

contracts/factory.sol:L403-L541

```
function createERC721DAO(
    ...
    address[] calldata _depositTokenAddress,
) external payable {
    checkCreateFeesSent(_depositChainIds);

    ...
    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),
                _ownerFeePerDepositPercent,
                _depositTime,
                _quorumPercent,
                _thresholdPercent,
                _safeThreshold,
                _depositChainIds,
                _daoAddress,
                _depositTokenAddress[i + 1],
                _admins,
                _onlyAllowWhitelist,
                _merkleRoot,
                0
            );
            ICommLayer(commLayer).sendMsg{
                value: msg.value / _depositChainIds.length - 1
            }(
                commLayer,
                _payload,
                abi.encode(_depositChainIds[i + 1], msg.sender)
            );
            unchecked {
                ++i;
            }
        }

    }
}
```

The highlighted code above ignores the fees for local chain deployment and incorrectly assigns all of the user's msg.value to other chains.

It will cause the contract to lose a share of createFee (msg.value / _depositChainIds.length) that it should have received.

Recommendation

Consider properly assigning createFee. for example, the above code can be modified like the following:

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

Status

The team has resolved this issue in commit [2b0ab218](2b0ab218).

9. Possible dos when creating dao

Severity: Low	Category: Denial of Service
Target: - contracts/Deployer.sol	

Description

In the StatationX protocol, while creating a dao, the deploySAFE() function is called to create a safe account for that dao.

contracts/Deployer.sol.sol:L85-L107

```
function deploySAFE(
    address[] calldata _admins,
    uint256 _safeThreshold,
    address _daoAddress
) external returns (address SAFE) {
    ...
    SAFE = ISafe(safe).createProxyWithNonce(
        singleton,
        _initializer,
        uint256(uint160(_daoAddress))
    );
}
```

In the above code, daoAddress is used as nonce to create safe account. However, if the nonce has already been used, it may cause the creation to revert.

This means that a malicious user can perform a front-run attack on the createDao transaction, causing create to be dosed.

Attach Scenario
a) User calls createERC20DAO at Factory.
b) Attacker grab data from calldata in mempool, calculate daoAddress in advance
c) Attacker frontrun user tx and calls deploySAFE at Deployer and deploys Safe with users data
d) User tx reverted as a result user can't create DAO

Recommendation

Consider catching the exception and trying to re-fetch the random number as a nonce to try to redeploy Safe. For example, it could be rewritten as follows:

```
function deploySAFE(
    address[] calldata _admins,
    uint256 _safeThreshold,
    address _daoAddress
) external returns (address SAFE) {
    bytes memory _initializer = abi.encodeWithSignature(
```

```
        "setup(address[],uint256,address,bytes,address,address,uint256,address)",
        _admins,
        _safeThreshold,
        0x0000000000000000000000000000000000000000,
        "0x",
        0xf48f2B2d2a534e402487b3ee7C18c33Aec0Fe5e4,
        0x0000000000000000000000000000000000000000,
        0,
        0x0000000000000000000000000000000000000000
    );

    uint256 nonce = getNonce(_daoAddress);

    do {
        try
            ISafe(safe).createProxyWithNonce(singleton, _initializer, nonce)
        returns (address _deployedSafe) {
            SAFE = _deployedSafe;
        } catch Error(string memory reason) {
            if (keccak256(bytes(reason)) != _SAFE_CREATION_FAILURE_REASON) {
                revert SafeProxyCreationFailed();
            }
            emit SafeProxyCreationFailure(gnosisSafeSingleton, nonce, _initializer);
            nonce = getNonce(nonce);
        } catch {
            revert SafeProxyCreationFailed();
        }
    } while (SAFE == address(0));
}

function getNonce(uint256 salt) internal view returns(uint256) {
    return uint256(keccak256(abi.encodePacked(address(this), block.number, salt)));
}
```

Status

The team has resolved this issue in commit [5c09785](5c09785).

10. Use call instead of transfer for native tokens transfer

Severity: Low	Category: Business logic
Target: - contracts/factory.sol	

Description

The transfer function is not recommended for sending native tokens due to its 2300 gas unit limit which may not work with smart contract wallets or multi-sig. Instead, call can be used to circumvent the gas limit.

contracts/factory.sol:L794-L803;L854-L863;L1269-L1277

```
function _buyGovernanceTokenERC20DAO() private {
    ...
    payable(
        _daoDetails.assetsStoredOnGnosis
            ? _daoDetails.gnosisAddress
            : _daoAddress
    ).transfer(_totalAmount);
    payable(
        ccDetails[_daoAddress].ownerAddress != address(0)
            ? ccDetails[_daoAddress].ownerAddress
            : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress
    ).transfer(ownerShare);
    ...
}

function _buyGovernanceTokenERC721DAO() private {
    ...
    payable(
        _daoDetails.assetsStoredOnGnosis
            ? _daoDetails.gnosisAddress
            : _daoAddress
    ).transfer(_totalAmount);
    payable(
        ccDetails[_daoAddress].ownerAddress != address(0)
            ? ccDetails[_daoAddress].ownerAddress
            : IERC20DAO(_daoAddress).getERC20DAOdetails().ownerAddress
    ).transfer(ownerShare);
    ...
}

function rescueFunds(address tokenAddr) external onlyOwner {
    uint256 balance = address(this).balance;
    payable(_owner).transfer(balance);
}
```

Recommendation

Consider using call instead of transfer for sending native token.

Status

The team has resolved this issue in commit [e140ccf6](e140ccf6).

11. Key events can be triggered arbitrarily

Severity: Low	Category: Logging
Target: - contracts/erc20dao.sol - contracts/erc721dao.sol	

Description

contracts/erc20dao.sol:L292-L298; contracts/erc721dao.sol:L372-L378

```
function emitSignerChanged(
    address _dao,
    address _signer,
    bool _isAdded
) external {
    Emitter(emitterContractAddress).changedSigners(_dao, _signer, _isAdded);
}
```

The lack of privilege control on emitSignerChanged() causes the ChangedSigners event to be able to be triggered by malicious users.

This can be misinterpreted by administrators or users who are monitoring the protocol off-chain , and can result in risks such as a reduction in the confidence of the protocol.

Recommendation

Consider designing appropriate access control for this function.

Status

The team has resolved this issue in commit 2b0ab218.

12. Incorrect boundary checking

Severity: Low	Category: Data Validation
Target: - contracts/erc721dao.sol	

Description

contracts/erc721dao.sol:L74-L104

```
function mintToken(
    address _to,
    string calldata _tokenURI,
    uint256 _amount
) public onlyFactory(factoryAddress) {
    if (balanceOf(_to) + _amount > erc721DaoDetails.maxTokensPerUser)
        revert MaxTokensMintedForUser(_to);
    if (!erc721DaoDetails.isNftTotalSupplyUnlimited) {
        require(
            Factory(factoryAddress)
                .getDAOdetails(address(this))
                .distributionAmount >= _tokenIdTracker,
            "Max supply reached"
        );
    }
    for (uint256 i; i < _amount; ) {
        _tokenIdTracker += 1;
        _safeMint(_to, _tokenIdTracker);
        _setTokenURI(_tokenIdTracker, _tokenURI);
        unchecked {
            ++i;
        }
    }
}
```

The above highlighted code wants to make sure that the total distribution of NFTs does not exceed max supply. but it ignores that this call also mints the _amount number of NFTs.

This can lead to the possibility that the total distribution of erc721Dao may exceed the expected amount.

Recommendation

Consider implementing proper boundary value checking, for example the above code could be modified like the following:

```
require(
    Factory(factoryAddress)
        .getDAOdetails(address(this))
        .distributionAmount >= _tokenIdTracker + _amount,
   "Max supply reached"
);
```

Status

The team has resolved this issue in commit 2b0ab218.

2.3 Informational Findings

13. Implementation contract could be initialized by everyone

Severity: Informational	Category: Configuration
Target: - contracts/erc721dao.sol - contracts/erc20dao.sol	

Description

According to OpenZeppelin, the implementation contract should not be left uninitialized.

An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. There is nothing preventing the attacker from calling the initializeERC20()/initializeERC721() function in erc20dao/erc721dao's implementation contract.

Recommendation

To prevent the implementation contract from being used, consider invoking the _disableInitializers function in the constructor of the erc20dao contract to automatically lock it when it is deployed.

Status

The team has resolved this issue in commit 2b0ab218.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 5ac0b43:

File	SHA-1 hash
Deployer.sol	975b43983b0e9db1fd87987b8281bec2c8c9f5bf
emitter.sol	994f76c55eba8d6be33df131e86e8dd3e19812aa
erc20dao.sol	63cf59c62573fb0d8d8502fa2c1a6026769155d8
erc721dao.sol	b364b22ec00f2c09391e034c884968df807e8317
factory.sol	5db7ab2315e10d74b6a041bf6981a39891fe7450
helper.sol	b42f45f05d49ec1b9635271a9fb25d5609221a94
proxy.sol	5854dcbdf11732568c53b6882442c4f25ef9e880
zairdrop.sol	dc6ed0fdb847004bcbca9de91be40e52c2446764
LayerZeroDeployer.sol	ffe73d6ba147a122c3deab679512c5842afe43fa
LayerZeroImpl.sol	f78d4adf384fb0dc5b56b35f2165b5f542e4fd7c
ExcessivelySafeCall.sol	42db88ccc2c140f37320238c60bcc2064a0edaea
ZetaImpl.sol	bacdf155717b60c90b33cdc3cc19138f446997d1
ZetaInteractor.sol	f238c4e63fb59e2b4cdda76e392cc434f5a14daf