SALUS SECURITY

SEP 2025

CODE SECURITY ASSESSMENT

XONE

Overview

Project Summary

- Name: XONE - DAO
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/hello-xone/xone_dao_contracts
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	XONE - DAO
Version	v3
Type	Solidity
Dates	Sep 19 2025
Logs	Sep 02 2025, Sep 15 2025, Sep 19 2025

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	6
Total Low-Severity issues	3
Total informational issues	3
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	Missing access control in _authorizeUpgrade	High	Business Logic	Resolved
2	Unfunctional unstakeAll	Medium	Business Logic	Resolved
3	Users may fail to unlock	Medium	Business Logic	Resolved
4	Incorrect clear for lockedAmounts	Medium	Business Logic	Resolved
5	Lock freeze can be bypassed	Medium	Business Logic	Acknowledged
6	Users can vote for one vote lock	Medium	Business Logic	Resolved
7	Centralization risk	Medium	Centralization	Acknowledged
8	Implementation contracts lack of _disableInitializers()	Low	Business Logic	Resolved
9	claimxXOC ignores input parameter and claims all rewards	Low	Business Logic	Acknowledged
10	Use call instead of transfer for native tokens transfer	Low	Business Logic	Resolved
11	Use of floating pragma	Informational	Configuration	Resolved
12	Missing two-step transfer ownership pattern	Informational	Business Logic	Resolved
13	Gas optimization suggestions	Informational	Gas Optimization	Resolved

2.2 Notable Findings

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

1. Missing access control in _authorizeUpgrade

Severity: High	Category: Business Logic
Target: - contracts/dao/DAOLock.sol	

Description

`DAOLockV` is the implementation contract behind a UUPS proxy. In this setup, the contract owner should be able to upgrade the implementation using `upgradeToAndCall`. However, the current implementation lacks access control for the `_authorizeUpgrade` function. As a result, anyone could trigger an upgrade, which is a significant security vulnerability.

contracts/dao/DAOLock.sol:L98-L101

```
function _authorizeUpgrade(
    address newImplementation
) internal virtual override {}
```

Recommendation

Add `onlyOwner` modifier to function `_authorizeUpgrade`

Status

The team has resolved this issue in commit 94a44c6.

2. Unfunctional unstakeAll

Severity: Medium	Category: Business Logic
Target: - contracts/stake/XOCStaking.sol	

Description

`XOCStaking` includes a function called `unstakeAll` that allows users to withdraw all of their assets at once. Internally, this function invokes `this.unstake`, which means the `msg.sender` seen by the `unstake` function will be `XOCStaking` itself—not the original user who called `unstakeAll`. This behavior breaks correct authorization logic and poses a serious security risk.

contracts/dao/DAOLock.sol:L98-L101

```
function unstakeAll(uint256 stakeIndex) external {
    this.unstake(stakeIndex, 0);
}

function unstake(uint256 stakeIndex, uint256 amount) public {
   require(
       stakeIndex < userStakes[_msgSender()].length,
       "Invalid stake index"
    );
    ...
}
```

Recommendation

Call `unstake` directly in the function `unstakeAll`.

Status

The team has resolved this issue in commit 94a44c6.

3. Users may fail to unlock

Severity: Medium	Category: Business Logic
Target: - contracts/stake/DAOLock.sol	

Description

In `DAOLock`, when a user unlocks their lock, the contract removes the lock by iterating over the entire `_candidateLocks` array to locate and delete the entry. However, this approach is vulnerable to malicious behavior: an attacker could inflate the `_candidateLocks` array by voting repeatedly, causing thousands of entries. As a result, when a user attempts to unlock, the transaction is likely to run out of gas and revert.

contracts/dao/DAOLock.sol:L208-L226

```
function unlock(
    uint256 lockIndex,
    uint256 amount
) public {
    ...
    if (lockInfo.lockType == LockType.VOTE && lockInfo.amount == 0) {
       _removeLockFromCandidate(lockInfo.candidateLockIndex, lockIndex);
     }
}
```

contracts/dao/DAOLock.sol:L256-L265

```
function _removeLockFromCandidate(uint256 candidateLockIndex, uint256 lockIndex)
internal {
    uint256[] storage _candidateLocks = candidateLocks[candidateLockIndex];
    for (uint256 i = 0; i < _candidateLocks.length; i++) {
        if (_candidateLocks[i] == lockIndex) {
            _candidateLocks[i] = _candidateLocks[_candidateLocks.length - 1];
            _candidateLocks.pop();
            break;
        }
     }
}
```

Recommendation

Use a `minVoteAmount` to mitigate the dos.

Status

The team has resolved this issue in commit 66bf400.

4. Incorrect clear for lockedAmounts

Severity: Medium	Category: Business Logic
Target: - contracts/stake/DAOLock.sol	

Description

In the `DAOLockV` contract, users can unlock their vote (i.e., cancel their candidacy) via the `unlock` function. When this happens, the contract resets the `lockedAmount` for the current term. However, there is a flaw: a user might have been a candidate in a previous term as well. If they call `unlock` for an older candidate request, the reset logic will inadvertently clear the `lockedAmount` for the current term instead. This could lead to incorrect state updates.

contracts/dao/DAOLock.sol:L241-L250

```
function unlock(
    uint256 lockIndex,
    uint256 amount
) public {
    if (lockInfo.lockType == LockType.CANDIDATE) {
        // must unlock all amount for candidate.
        require(lockInfo.amount == 0," Must unlock full amount for candidate lock");
        _unlockCandidateVotes(lockIndex);

        lockedAmounts[_msgSender()][currentTerm] = 0;
    }
}
```

Recommendation

Clear the correct term's `lockedAmounts`.

Status

The team has resolved this issue in commit 94a44c6.

5. Lock freeze can be bypassed

Severity: Medium	Category: Business Logic
Target: - contracts/stake/DAOLock.sol	

Description

In the `DAOLock` contract, users' locked votes can be frozen by the owner. Upon calling the `unlock` function, the contract unfreezes all vote locks across the board. This introduces a critical flaw: if a user has a vote that was previously frozen, the `unlock` function can still bypass the freeze and unfreeze that lock. This behavior undermines the intent of the freeze feature and compromises the security intended by the owner.

contracts/dao/DAOLock.sol:L98-L101

```
function _unlockCandidateVotes(uint256 candidateLockIndex) internal {
    uint256[] memory voteLocks = candidateLocks[candidateLockIndex];

    // Loop all votes for this candidate index.
    // Unlock for users.
    for (uint256 i = 0; i < voteLocks.length; i++) {
        ...
    }
}
```

Recommendation

Refactor the freeze logic.

Status

This issue has been acknowledged by the team. The team stated that according to business design, there will be no frozen assets.

6. Users can vote for one vote lock

Severity: Medium	Category: Business Logic
Target: - contracts/stake/XOCStaking.sol	

Description

In the `DAOLock` contract, users can cast votes for a candidate. The `voteForCandidate` function currently includes a check `candidate != address(0)` to ensure that votes are not directed to the zero address. However, this check is insufficient: even if the `candidate` field in a valid vote lock isn't `address(0)`, it doesn't guarantee that the address corresponds to an actual candidate. As a result, users may be able to cast votes for non-existent or invalid candidates, which undermines the intended functionality.

contracts/dao/DAOLock.sol:L159-L180

```
function voteForCandidate(
    uint256 xXOCAmount,
    uint256 candidateLockIndex
) external{
    ...
    LockInfo storage candidateLockInfo = locks[candidateLockIndex];
    // Make sure that the candidate lock index is one valid index.
    require(candidateLockInfo.candidate != address(0), "Invalid candidate address");

    LockInfo memory lockInfo = LockInfo(
        _msgSender(),
        candidateLockInfo.candidate,
        LockType.VOTE, // Lock type is VOTE
        xXOCAmount,
        candidateLockIndex,
        false // default is false. Not freeze.
    );
}
```

Recommendation

Enhance the safety check and disallow to vote for one non-candidate lock.

Status

The team has resolved this issue in commit 94a44c6.

7. Centralization risk

Severity: Medium	Category: Centralization
Target: - contracts/stake/XOCStaking.sol	

Description

In Xone contracts, there exists one privileged role, `OWNER`. This role has the authority to execute some key functions such as `emergencyWithdra`.

If these roles' private keys are compromised, an attacker could trigger these functions to steal all funds.

contracts/dao/DAOLock.sol:L98-L101

```
function emergencyWithdraw() external onlyOwner {
     payable(owner()).transfer(address(this).balance);
}
```

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

Status

This issue has been acknowledged by the team.

8. Implementation contracts lack of _disableInitializers()

Severity: Low	Category: Configuration

Target:
- contracts/dao/DAOLock.sol
- contracts/stake/StakingReward.sol
- contracts/stake/XOCStaking.sol

Description

The `DAOLock` , `StakingReward` and `XOCStaking` contracts do not call OpenZeppelin's `_disableInitializers()` in its constructor.

As a result, anyone can send a direct transaction to the implementation address (not the proxy) and invoke `initialize()`. Gaining owner permissions within the implementation's own storage context on all contracts. Additionally, the corresponding `xXOCToken` can also be changed within both the `StakingReward` and `XOCStaking` contracts.

Recommendation

Add a constructor that locks the implementation:

```
constructor() {
    _disableInitializers();
}
```

Status

The team has resolved this issue in commit 94a44c6.

9. claimxXOC ignores input parameter and claims all rewards

Severity: Low	Category: Business Logic
Target: - contracts/stake/XOCStaking.sol	

Description

The `claimxXOC(uint256[] stakeIndexes)` function accepts an array of stake indexes, but the parameter is never used in the logic. Instead, the function simply transfers the entire `userxXOCBalance[msg.sender]` to the user, resetting it to zero. This behavior contradicts the function signature and comment, and the emitted event misleadingly includes the unused stakeIndexes.

contracts/dao/XOCStaking.sol:L261-L271

```
function claimxXOC(uint256[] calldata stakeIndexes) external {
    uint256 totalxXOCToClaim = userxXOCBalance[_msgSender()];
    userxXOCBalance[_msgSender()] = 0;

    require(totalxXOCToClaim > 0, "No xXOC to claim");

    // Mint xXOC tokens to user
    xXOCToken.mint(_msgSender(), totalxXOCToClaim);

    emit xXOCClaimed(_msgSender(), stakeIndexes, totalxXOCToClaim);
}
```

Recommendation

Remove the unused parameter, or correctly implement per-stake claiming.

Status

This issue has been acknowledged by the team. The team stated that the index is mainly used for backend statistical data.

10. Use call instead of transfer for native tokens transfer

Severity: Low	Category: Business logic
Target: - contracts/stake/StakingReward.sol - contracts/stake/XOCStaking.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/stake/StakingReward.sol:L51,L58

```
payable(_msgSender()).transfer(amount);
payable(owner()).transfer(amount);
```

contracts/stake/XOCStaking.sol:L214,L321

```
payable(_msgSender()).transfer(unstakeAmount);
payable(owner()).transfer(address(this).balance);
```

Recommendation

Consider using call instead of transfer for sending native token.

Status

The team has resolved this issue in commit 94a44c6.

2.3 Informational Findings

11. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity >=0.8.0;
```

The Rainlink uses a floating compiler version >=0.8.0.

Using a floating pragma >=0.8.20 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has resolved this issue in commit 94a44c6.

12. Missing two-step transfer ownership pattern

Severity: Informational	Category: Business logic
Target: - contracts/stake/XOCStaking.sol	

Description

The `XOCStaking` contract uses a custom function `transferOwnership()` which is a simple mechanism to transfer the ownership not supporting a two-step transfer ownership pattern. This simpler mechanism can be useful for quick tests, but projects with production concerns are likely to outgrow it. Transferring ownership is a critical operation and this could lead to transferring it to an inaccessible wallet or renouncing the ownership, e.g. mistakenly.

contracts/stake/XOCStaking.sol:L328-L331

```
function transferOwnership(address newOwner) public override onlyOwner {
    require(newOwner != address(0), "Invalid new owner address");
    _transferOwnership(newOwner);
}
```

Recommendation

Consider using the Ownable2Step contract from OpenZeppelin instead.

Status

The team has resolved this issue in commit 94a44c6.

13. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- contracts/dao/DAOLock.sol
- contracts/stake/StakingReward.sol
- contracts/stake/XOCStaking.sol

Description

Finding 1: Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. Cache array length outside of the loop can save gas.

contracts/dao/DAOLock.sol:L192, L246,L264,L298,L340,L355,L364

```
for (uint256 i = 0; i < lockIndexes.length; i++) {
for (uint256 i = 0; i < _candidateLocks.length; i++) {
for (uint256 i = 0; i < voteLocks.length; i++) {
for (uint256 i = 0; i < unlockIndexes.length; i++) {
for (uint256 i = 0; i < userLocks[user].length; i++) {
for (uint256 i = 0; i < freezeLockIndexes.length; i++) {
for (uint256 i = 0; i < unfreezeLockIndexes.length; i++) {
```

contracts/stake/StakingReward.sol:L38

```
for (uint256 i = 0; i < users.length; i++) {
```

contracts/stake/StakingReward.sol:L124,L233

```
for (uint256 i = 0; i < tierIndexes.length; i++) {
for (uint256 i = 0; i < stakeIndexes.length; i++) {
```

Recommendation

Consider using the above suggestions to save gas.

Status

The team has resolved this issue in commit 94a44c6.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 53a0ad5:

File	SHA-1 hash
DAOLock.sol	76d40fb876ae8ad8aab5fafdfcd183ca100274ab
StakingReward.sol	7e26cfe4344a142616b24f6c311b517354b01546
XOCStaking.sol	f9b6e9500d30334a02a80749b1165519d641f225
xXOC.sol	c93b71fb0593ac8aa5837ccc1259fdeb78cfee65