



# Smart Contract Security Assessment Report For

stack

**Audited by**

Deliriusz  
Martin Petrov

**Securr**

<https://securr.tech>  
[suhrad@securr.tech](mailto:suhrad@securr.tech)

# Table of Content

## 1. Disclaimer

## 2. Severity Definitions

## 3. Scope

## 4. Issues found

## 5. Vulnerabilities

- [C-1] Lack of effective slippage protection leads to users losing all XCT tokens during swap
- [C-2] Anyone is able to steal all XCT tokens from other users who approved the transfer in SubscriptionBalance
- [C-3] Unprotected AppNFT.mint() function allows anyone to mint NFT for free
- [H-1] AppNFT transfer DoS via granting roles to too much users
- [H-2] No negative balance check when updating balance allows apps to run for free
- [H-3] Cluster owner's STACK locked lost when trying to withdraw it for a delisted subnet
- [H-4] Part of subnet revenue lost when new cluster is joining
- [M-1] Hardcoded block.timestamp for token swap may be unfavourable to the user during adverse market conditions
- [M-2] Usage of deprecated transfer to send eth
- [M-3] Wrong slippage upper-bound
- [M-4] Dust is left in the smart contract when distributing revenue to clusters
- [M-5] Pausing Centralization Vulnerability
- [M-6] Missing credit expiry checks leads to unexpected results



# Table of Content

- [M-7] DoSing revenue distribution when too much clusters is added
- [M-8] SubnetCluster deployment can be frontruned and DoSed or exploited
- [M-9] Removing DarkMatterNFT from list of supported NFTs locks it for the users that did not manage to withdraw them
- [M-10] Cluster revenue may get lost then withdrawing cluster for delisted subnet
- [M-11] Possible role exploit using reentrancy vector in AppNFT
- [M-12] Gas griefing and DoS app resources change
- [M-13] The same app may be deleted multiple times, breaking protocol invariants
- [M-14] Withdrawing misbehaving operator NFT does not stop the revenue generation
- [M-15] Delisting a subnet does not stop revenue streaming for users using it
- [M-16] Not using safe operations may make XCTMinter unfunctional after stablecoin address change
- [M-17] Admin can make XCTMinter disfunctional by changing or withdrawing stablecoin
- [M-18] Missing tests
- [M-19] Clusters are in risk of not getting paid and the funds being stuck in SubnetDAODistributor when setting revenue is performed often
- [L-1] \_safeMint creates reentrancy attack possibility



# Table of Content

- [L-2] Wrong time check
- [L-3] Not blocking initialization in constructor leads to unexpected initialization of logic contract by anyone
- [L-4] Sending data with ERC721 and ERC1155 token transfers may lead to unexpected results
- [L-5] Wrong storage gap value may break storage layout in new contract version
- [L-6] Checks effects interactions pattern not followed
- [L-7] No check for ERC20 transfer return value
- [L-8] Subnet may have more active clusters than maxClusters
- [L-9] Replicating the same storage layout for diamond proxy may break the protocol after updating
- [L-10] Implemented cluster signup logic does not conform to requirements
- [L-11] Gas griefing and DoS when subnet adds too much of whitelisted addresses
- [L-12] Missing events emits in important functions
- [L-13] Misleading use of `safeMint()` in `AppNFT.mint()` may lead to misuse of the NFT
- [L-14] Wrong calculations of app subnets in `ContractBasedDeployment.getSubnetsOfApp()`
- [L-15] Allowing arbitrary `UniswapV2Router` address may lead to user funds being lost
- [I-1] Remove pausable design pattern reimplementing
- [I-2] Use a more recent version of solidity consistently across the whole codebase
- [I-3] Usage of `SafeMath` is deprecated in Solidity  $\geq v0.8.0$
- [I-4] Duplicated `Counters` library definition
- [I-5] Unnecessary address casting
- [I-6] Remove commented functionalities
- [I-7] All contracts are unlicensed
- [I-8] Missing `NatSpec`
- [I-9] Unclear `TestERC20/STACK` implementation



# Disclaimer

Please note that this audit report is provided "as is", without any representations or warranties of any kind. The author and their employer assume no liability for any damages arising from or in connection with the contents of this report.

The copyright of this report solely belongs to the author. Any reproduction or distribution of this report without the explicit permission of the author is strictly prohibited.



# Severity Definitions

## Impact:

- High: Results in a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium: Involves a small amount of fund loss (such as value leakage) or affects a core functionality of the protocol.
- Low: Leads to unexpected behavior with some of the protocol's functionalities that are not critical.

## Likelihood:

- High: The attack path is possible with reasonable assumptions that replicate on-chain conditions, and the cost of the attack is relatively low compared to the potential amount of funds that can be stolen or lost.
- Medium: The attack vector is conditionally incentivized, but still relatively likely.
- Low: Requires numerous or highly unlikely assumptions or demands a significant stake from the attacker with little to no incentive.

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



# Scope

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/DarkMatterNFT.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/LinkNFTs.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/MultiAccessControlUpgradeable.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterA.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterProxy.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetDAODistributor.sol>



# Scope

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/Subscription.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TestERC20.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalanceCalculator.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TokensRecoverable.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TokensRecoverableOwner.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTERC20.sol>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol>





# Issues found

Severity	Count
Critical risk	3
High risk	4
Medium risk	19
Low risk	15
Informational	9
Recommendations	5



# [C-1] Lack of effective slippage protection leads to users loosing all XCT tokens during swap

**Current Status:**  
**Fixed**

**Context:**

<https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L200-L214>

**Description:**

When performing a swap in `XCTMinter.easyBuyXCT()` and `XCTMinter.buyXCT()`, slippage parameter is passed, however assuring it is not sufficient. The root of the problem lies in calculating resulting amount using UniswapV2 spot price:

```
uint256[] memory amounts = routerV2.getAmountsOut(
    msg.value,
    path
);
```

Spot price is easily manipulable in UniswapV2 and can be driven to dust amounts during sandwich attack. They are, however, used in calculating `minAmountOut` when invoking `routerV2.swapExactETHForTokensSupportingFeeOnTransferTokens`:



# [C-1] Lack of effective slippage protection leads to users loosing all XCT tokens during swap

```
uint prevUSDC = StableCoinToken.balanceOf(address(this));
routerV2.swapExactETHForTokensSupportingFeeOnTransferTokens{
    value: msg.value
}(
    amounts[1].mul(slippageFactor).div(100), // @audit operating on manipulable `amounts` returned from
UniswapV2
    path,
    address(this),
    block.timestamp
);

.....
usdcReceived = StableCoinToken.balanceOf(address(this)).sub(prevUSDC);

uint xctToMint = usdcReceived.mul(stepUpFactor);
XCTToken.mint(msg.sender, xctToMint); // @audit msg.sender can receive dust amounts due to sandwich
attack
```

Assuming that the price is manipulated, spot price will only guarantee that the user receives at minimum manipulated price less slippage. This effectively means up to 100% losses for the users.

## Recommended Mitigation Steps:

Either use TWAP onchain oracle to calculate minimum amount of tokens received based on multiple blocks, or use VWAP oracle like Chainlink for the same.

## Mitigation:

Team resolved the issue by adding deadline and minAmountOut parameters for a swap and tested it's validness creating test case XM3.

Implemented fix resolves the issue. One fact to note is that now the smart contract caller is responsible for calculating acceptable price slippage.



# [C-2] Anyone is able to steal all XCT tokens from other users who approved the transfer in SubscriptionBalance

**Current Status:**

Fixed

**Context:**

<https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L202-L227>

**Description:**

`SubscriptionBalance.addBalance()` is a public function called by anyone and allows `nftOwner` address to add XCT balance for specific NFT id account. There is no check however, if `nftOwner` passed is: (1) `msg.sender`, (2) is owner of the NFT. This allows anyone to drain XCT tokens up to the max user allowance, because `nftOwner` that the tokens are taken from may be arbitrary address:

```
function addBalance(address nftOwner, uint256 nftID, uint256 balanceToAdd)
    public
    whenNotPaused
{
    updateBalance(nftID);
    _addBalance(nftOwner, nftID, balanceToAdd);
}

function _addBalance(address sender, uint256 nftID, uint256 _balanceToAdd)
    internal
{
    XCTToken.transferFrom(
        sender, // @audit `sender` is `nftOwner`, that can be any arbitrary address
        address(this),
        _balanceToAdd
    );
    .....
}
```



## [C-2] Anyone is able to steal all XCT tokens from other users who approved the transfer in SubscriptionBalance

This allows anyone to drain the funds using `SubscriptionBalance.withdrawAllOwnerBalance()`, to get the XCT tokens back.

### Recommended Mitigation Steps:

Perform checks similar to `withdrawAllOwnerBalance()`, to verify that XCT can be deducted from `msg.sender` only.

### Mitigation:

Team resolved the issue by adding `deadline` and `minAmountOut` parameters for a swap and tested its validity creating test case XM3.

Implemented fix resolves the issue. One fact to note is that now the smart contract caller is responsible for calculating acceptable price slippage.



## [C-3] Unprotected AppNFT.mint() function allows anyone to mint NFT for free

**Current Status:**  
Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol#L54-L60>

**Description:**

AppNFT is meant to be used as a mean to access app listed in StackOS marketplace. Because of that, the app creator has to protect the minting, to not let anyone get the app for free. In current implementation however, the mint() function is not protected, allowing anyone to free mint the NFT. Because AppNFTs are meant to have a value, a malicious actor can free mint the NFTs and sell them, driving the price to 0, and stealing from app creator.

```
function mint(address to)
external
{
    uint tokenID = nextTokenID.current();
    _safeMint(to, tokenID);
    nextTokenID.increment();
}
```

**Recommended Mitigation Steps:**

Protect mint function by one of the roles defined in the smart contract. Grant roles in the initializer function.



## **[C-3] Unprotected AppNFT.mint() function allows anyone to mint NFT for free**

### **Mitigation:**

Team resolved the issue and tested it's validness creating test case SD48.

Implemented fix resolves the issue.



# [H-1] AppNFT transfer DoS via granting roles to too much users

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/MultiAccessControlUpgradeable.sol#L246>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/MultiAccessControlUpgradeable.sol#L299-L322>

**Description:**

According to the docs, AppNFT owners can create their own roles and assign them to NFTs. Even though StackOS only requires finite amount of them, the owners are free to add as many roles as they like. They are also free to add as many members to the role. In case that an app decides to grant on-chain roles to all their users, community managers, or any larger group of people, the amount of users of a group will raise significantly. This DoSes transfers, because it internally uses `MultiAccessControlUpgradeable._removeAllRoles()`, which goes over all groups and its members in a loop and deletes them:





# [H-1] AppNFT transfer DoS via granting roles to too much users

```
function _removeAllRoles(uint256 nftID)
    internal
    virtual
    {
        for(uint256 i = 0; i < nftRoleList[nftID].length; i++)
        {
            bytes32 role = nftRoleList[nftID][i];
            address[] memory memberList = nftToAccountRoles[nftID][role].memberList;
            for(uint256 j = 0; j < memberList.length; j++)
            {
                address member = memberList[j];

                if(!nftToAccountRoles[nftID][role].memberMap[member].active)
                    continue;

                uint256 userRoleID = nftToAccountRoles[nftID][role].memberMap[member].userRoleID;

                delete accountToNFTRoles[member][userRoleID];
            }

            delete nftToAccountRoles[nftID][role];
        }
    }
}
```

This makes AppNFT untransferrable, which impacts the app owners significantly.

## Recommended Mitigation Steps:

Use versioned approval pattern, which removes all permissions in a constant  $O(1)$  time. It works in this way:

- Roles are versioned. There is one global version kept for all of them internally, e.g.: `roles[version][roleId][member]`
- Each time that all roles are revoked, the version is increased by 1. This invalidates all the roles using single SSTORE operation:  
`version = version + 1`
- Now, because `version` changed, all calls for `hasRole(roleId, member)` will be cleared, because the mapping in new version is not set yet.

Real life example, is using `vaultId` in DelegateCash:

<https://github.com/delegatecash/delegate-registry/blob/df13cec0a57bcfc6558d523f6ddfbcb0becb5a78a/src/DelegationRegistry.sol>



# [H-1] AppNFT transfer DoS via granting roles to too much users

## **Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [H-2] No negative balance check when updating balance allows apps to run for free

Current Status:  
Acknowledged

Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L428-L478>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L559-L577>

Description:

A user can have the underwater part of their balance remitted, because `SubscriptionBalance.updateBalance()` function gets user `balances - costs` from `SubscriptionBalance.calculateUpdatedPrevBal()`, where it doesn't check if the position is underwater:

```
function calculateUpdatedPrevBal(
    uint256 totalCostIncurred,
    uint256[3] memory prevBalance
)
internal
pure
returns (uint256[3] memory prevBalanceUpdated)
{
    uint temp;
    uint temp2;
    for(uint i = 0; i < prevBalance.length; i++)
    {
        temp = prevBalance[i];
        temp2 = min(temp, totalCostIncurred); // @audit in case that the last (third) previous balance
does not suffice,
        prevBalanceUpdated[i] = temp.sub(temp2); // totalCostIncurred will be > 0, and still
`prevBalanceUpdated` will not
        totalCostIncurred = totalCostIncurred.sub(temp2); // account for this
    }
}
```

This means, that the user can receive a free computing power when their position is userwater.



## [H-2] No negative balance check when updating balance allows apps to run for free

After the calculations are done, `lastBalanceUpdateTime` is updated, remitting all the debt they accrued until the function was called:

```
nftBalances[nftID].lastBalanceUpdateTime = block.timestamp;
```

### Recommended Mitigation Steps:

Because blockchain is asynchronous by nature, it will never be guaranteed that the balance update will always be executed on timely manner. That's why we recommend the following changes to the design of payments:

1. change `NFTBalance.previousBalances` from `uint256[3]` to `int256[3]` and change logic to allow for negative balance
2. when user account is under specific threshold, e.g. has funds for less than 24 hours, emit an event first and then shut down all user running app instances, until they top up their balance

### Mitigation:

The `dripRate` is the amount of deduction per second. The balance is deducted only in multiples of `dripRate`.

The clusters keep track of the time taken for the balance to reach less than one `dripRate`. When that happens the apps are killed.

Team will have off-chain measures to protect against free app utilization.



# [H-3] Cluster owner's STACK locked lost when trying to withdraw it for a delisted subnet

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L617>

**Description:**

When a subnet gets delisted, cluster owner can get back their locked STACK token, NFT, and delist a cluster as well. However, due to the wrong assignment, balance of locked STACK is always set to 0, making a cluster owner loose their whole stake. The tokens will be locked in the smart contract without a possibility to retrieve them.

```
uint256 bal = balanceOfStackLocked[walletAddress] = 0; // @audit bal is always 0, due to setting
balanceOfStackLocked[walletAddress] to 0 before
balanceOfStackLocked[walletAddress] = 0; // @audit this line is correct, given than the line above
correctly caches locked balance
subnetClusters[subnetId][clusterId].listed = 3; // delisted cluster as withdrawn

StackToken.transferFrom(address(this), _msgSender(), bal); // @audit owner always gets 0
```

In the line above, `balanceOfStackLocked[walletAddress] = 0` is evaluated first, setting `balanceOfStackLocked[walletAddress]` to 0. Then it's assigned to `bal`, which is later used as an amount to transfer back to the cluster owner.

**Recommended Mitigation Steps:**

Change line 617 from:

```
uint256 bal = balanceOfStackLocked[walletAddress] = 0;
```

to:

```
uint256 bal = balanceOfStackLocked[walletAddress];
```

This will set balance to send to currently locked value correctly.



## **[H-3] Cluster owner's STACK locked lost when trying to withdraw it for a delisted subnet**

### **Mitigation:**

Team resolved the issue as recommended and tested it's validness creating test cases SC8-T2 and SC8-T3

Implemented fix resolves the issue.



# [H-4] Part of subnet revenue lost when new cluster is joining

## Current Status:

Fixed

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L375-L384>

## Description:

When a new cluster joins a subnet and is whitelisted, its Status: is set to 2 and then cluster weight is set:

```
they signup
    if (whitelistedClusters[subnetId][i] == ownerAddress) {
        subnetClusters[subnetId][clusterId].listed = 2; // whitelisted clusters are approved as
        SubnetDAODistributor.setClusterWeight(
            subnetId,
            clusterId,
            DefaultWhitelistedClusterWeight
        );
        isWhitelisted = true;
        break;
    }
```

The problem is that setting a weight internally calls `SubnetDAODistributor.assignRevenues()`, which distributes accrued subnet revenue to all whitelisted clusters. And because `listed` Status: is already set to 2, the cluster will be eligible for the revenue inappropriately. Additionally, at this point, cluster's struct is not filled in, and every other property like cluster wallet address is 0 at this point, so the revenue will be accredited to 0x0 address, which means that it's lost.

## Recommended Mitigation Steps:

Call `SubnetDAODistributor.assignRevenues()` before setting listed Status: to 2



## [H-4] Part of subnet revenue lost when new cluster is joining

### **Mitigation:**

Team resolved the issue by changing by setting cluster as listed after calling `SubnetDAODistributor.setClusterWeight()` and tested it's validness creating test cases SD14 and SD15.

Implemented fix resolves the issue.





# [M-1] Hardcoded block.timestamp for token swap may be unfavourable to the user during adverse market conditions

**Current Status:**  
Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L207-L214>

**Description:**

UniswapV2 introduces swap staleness check, by using maximum `block.timestamp` as the latest moment for the swap to succeed. It's meant to be passed by transaction originator, and fail the transaction if it stays in the mempool for too long. However, in case of `XCTMinter.easyBuyXCT()` and `XCTMinter.buyXCT()`, `block.timestamp` is passed always.

```
routerV2.swapExactETHForTokensSupportingFeeOnTransferTokens{
  value: msg.value
}{
  amounts[1].mul(slippageFactor).div(100),
  path,
  address(this),
  block.timestamp // @audit hardcoded block.timestamp will always be effective
};
```

This exposes users to economic losses, if the transaction is executed long after sending it to mempool. At the time of execution, the price may change and current swap may become unfavourable to the user.

**Recommended Mitigation Steps:**

Add `deadline` input parameter marking max swap time to the `easyBuyXCT()` and `buyXCT()` functions. Pass it to Uniswap V2 swap as input parameter.



# **[M-1] Hardcoded block.timestamp for token swap may be unfavourable to the user during adverse market conditions**

## **Mitigation:**

Team resolved the issue as recommended and tested it's validness creating test case XM4.

Implemented fix resolves the issue.



# [M-2] Usage of deprecated transfer to send eth

## Current Status:

Fixed

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TokensRecoverable.sol#L27>

## Description:

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

1. The claimer smart contract does not implement a payable function.
2. The claimer smart contract does implement a payable fallback that uses more than 2300 gas units.
3. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.
4. Additionally, using higher than 2300 gas might be mandatory for some multi-sig wallets.
5. Even if the current design is for using an EOA admin account since this function can only be called by the admin, you might decide to use a multisig in a later stage, and using call() will work for both, unlike transfer().

## Recommended Mitigation Steps:

Use `call()` with value instead of `transfer()`

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



## [M-3] Wrong slippage upper-bound

### Current Status:

Fixed

### Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L253>

### Description:

A common value used for the slippage tolerance is 1%. However, currently in buyXCT 100% upper-bound was hardcoded which can result in an extremely large fund loss for the user in a specific edge-case situation in a highly volatile or illiquid market.

In easyBuyXCT by contrast there is no validation on the slipFactor input argument.

### Recommended Mitigation Steps:

Slippage minimum amount upper bound implementation is a trade-off between ensuring the transaction's success and accepting the potential for large slippage. It's recommended to monitor market conditions and adjust the slippage tolerance accordingly to strike the right balance for your specific situation. So the best solution would be a setter function that can be accessed only by the owner.

In addition, add the missing validation in the easyBuyXCT function while not breaking the consistency with buyXCT.

### Mitigation:

The slippage parameter is removed, so the upper bound check is removed.

Removing slippage parameter resolves this issue.



# [M-4] Dust is left in the smart contract when distributing revenue to clusters

**Current Status:**  
**Acknowledged**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetDAODistributor.sol#L110-L120>

**Description:**

When calling `SubnetDAODistributor.assignRevenues()`, total revenues are distributed to clusters, based on their weights. However, depending on the weights, some dust amount can be left in the smart contract, due to rounding issues. Because anyone can run it anytime, distribution may frequently accrue dust, leaving the contract with locked subnets funds.

**Recommended Mitigation Steps:**

Please add a variable tracking funds left to be distributed inside the cluster distribution loop, and assign the leftovers back to `subnetRevenue`.

**Mitigation:**

After doing the calculations, this seems insignificant.

Consider 1 second per block

each block, we accumulate dust of  $-10^{(-18)}$  dollars

To reach 1 dollar, we need 31,709,791,983.7 years

Team acknowledges this issue, pointing that it has very low economic impact in practice.



# [M-5] Pausing Centralization Vulnerability

**Current Status:**  
**Acknowledged**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L229>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L266>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L281>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L323>

**Description:**

It's a centralization vulnerability if an owner can pause not only inbound but outbound functionality. It's a best practice to be able to pause inbound (deposit) functionality but not outbound (withdraw) methods.

**Recommended Mitigation Steps:**

Remove the `whenNotPaused` modifier from the withdraw functions like `withdrawAllOwnerBalance`, `withdrawBalanceLinked`, `withdrawBalance`, and `_withdrawBalance` in `SubscriptionBalance.sol` and `sellXCT` in `XCTMinter.sol`.



## [M-5] Pausing Centralization Vulnerability

### **Mitigation:**

The pausable modifiers are kept for the withdraw functions, as it is possible that a malicious user might attempt to withdraw the funds he might have manipulated.

Team acknowledges the issue, pointing that it may stop possible attacker from withdrawing the assets.



# [M-6] Missing credit expiry checks leads to unexpected results

## Current Status:

Fixed

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L299-L336>

## Description:

Upon adding subscription balance as a credit is meant to be withdrawable by DAO after specific amount if time. This, however is not enforced on the smart contract level. The only check that is being done for the timestamp is:

```
function addBalanceAsCredit(
    address sender,
    uint256 nftID,
    uint256 _balanceToAdd,
    uint256 _expiryUnixTimestamp
)
...
require(
    creditsExpiry[sender][nftID].expiryTimestamp < _expiryUnixTimestamp,
    "Invalid credits expiry value"
);
```

This condition only checks if the `_expiryUnixTimestamp` is bigger than stored one, and does not check if it's in the past, or far in the future. And because it's meant to be withdrawable by DAO, it can be either withdrawn too soon, or the payer may maliciously pass huge expiry timestamp, making the credit effectively non-withdrawable by the DAO.

## Recommended Mitigation Steps:

Check if expiry is at least `block.timestamp + <predefined-period>` and limit it to an upper bound to a reasonable value.





## **[M-6] Missing credit expiry checks leads to unexpected results**

### **Mitigation:**

The changes are made as per recommendation. Now there is a min and max bound for the expiry time, so that the creditor cannot put an indefinite time.

Implemented fix resolves the issue.



# [M-7] DoSing revenue distribution when too much clusters is added

**Current Status:**  
**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetDAODistributor.sol#L111-L120>

**Description:**

When subnet revenues are distributed to clusters, first total amount of cluster is read, and then every cluster is checked for being active and if so, the revenues are split according to the weights. That means that the more clusters there are, the more operations have to be performed. Assigning revenue to 1 active cluster yields 3 **SLOADs**, 1 **SSTORE** per cluster. Given that most probably all of them will be cold, and most clusters current revenue will be 0, in worst case scenario every cluster costs **~30\_000 gas**. With around 500 clusters, the operation is no longer possible to perform, reaching 15M block. Additionally, in current implementation there is no way to remove a cluster from the cluster array, so it can only increase. Even inactive clusters add up to the gas cost.

**Recommended Mitigation Steps:**

Looping over unbounded loops should be avoided at all times, and keeping an internal accounting instead of on-chain calculations using loops should be preferred. Ideally, this should be fixed in the first place. However, this requires broader redesign in how the cluster ids are assigned.



## [M-7] DoSing revenue distribution when too much clusters is added

### Mitigation:

The logic has been changed. Now each cluster can get their revenue individually.

Implemented fix resolves the issue by keeping single only-growing revenue index for subnet and clusters synchronizing to it. There is a possibility of dust amounts being locked in the smart contract, given the following formula:  $(\text{newRevenue} * \text{weight}) / \text{MAX\_TOTAL\_WEIGHT}$  may leave small reminder for non fully divisible numbers. Given that only very small amounts of the token impacted, the team acknowledges it and is not planning to change it.



# [M-8] SubnetCluster deployment can be frontrun and DoSed or exploited

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/scripts/helper.js#L372-L429>

**Description:**

Deployment of SubnetCluster is a multistep operation, requiring to deploy SubnetClusterA, SubnetClusterB and SubnetClusterProxy, initializing it afterwards. The deployment script used does all of the above operations separately, waiting for each transaction to succeed. A malicious actor can frontrun SubnetClusterProxy initialization, initializing it maliciously. If it's caught by the team, the smart contracts would have to be redeployed, and the situation may repeat itself, effectively DoSing the deployment. Worst case scenario, if it's not being caught, a malicious actor can initialize the smart contract with all the same parameters, but different \_GlobalDAO, which will allow them to perform all admin actions on the subnets and clusters, e.g. stealing all DarkMatterNFTs after some time.

Additionally, SubnetClusterA is left uninitialized on its own, it's only initialized in the context of SubnetClusterProxy. While we were not able to identify any attack vector in case that a malicious actor initializes it manually, it's considered a bad practice.

**Recommended Mitigation Steps:**

We recommend to create a constructor for SubnetClusterProxy, which calls diamondCut() for all functions and initializes the proxy in a single transaction. This makes this operation atomic, making frontrunning and DoSing impossible.



## **[M-8] SubnetCluster deployment can be frontrun and DoSed or exploited**

### **Mitigation:**

Team resolved the issue by creating a constructor with call to SolidStateDiamond parent.. Calling parent constructor will set the owner of the contract, so only the owner can call diamondCut.

Implemented fix resolves the issue.



# [M-9] Removing DarkMatterNFT from list of supported NFTs locks it for the users that did not manage to withdraw them

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L692-L699>

**Description:**

`SubnetClusterB.removeNFTContract()` allows to remove NFT address that serves as the DarkMatterNFT. It does not check if all of the NFTs are already withdrawn, locking them in the contract until the contract will be added again. Locking them has direct economic impact to the users, as they are meant to be expensive, as they are required to create a subnets and clusters.

**Recommended Mitigation Steps:**

We recommend checking if all the NFTs are already withdrawn from the contract and if not, force withdraw those left to the owners when removing support for an NFT.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue. Because the function does not exist, there is no possibility to lock the NFTs.



# [M-10] Cluster revenue may get lost then withdrawing cluster for delisted subnet

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L597-L642>

**Description:**

Any change in cluster operation - weight change, listing, delisting, et al, should settle the cluster balance. It happens by calling `SubnetDAODistributor.setClusterWeighth()`, which internally settles clusters based on their weights.

If `SubnetClusterA.changeSubnetStatus:Listed()` is set to `false`, then either admin or owner can call `SubnetClusterB.withdrawClusterForDelistedSubnet()`. It effectively delists the cluster. However, because `SubnetDAODistributor` is not called here to set weights, the revenue calculations are not being done and the cluster owner does not receive due revenue.

**Recommended Mitigation Steps:**

When withdrawing cluster, please also set its weight to 0 in `SubnetClusterB.withdrawClusterForDelistedSubnet()` by calling `SubnetDAODistributor.setClusterWeight()`. This will internally settle accounting.

**Mitigation:**

Team resolved the issue as recommended. Now the cluster accounting is done before setting its weight to zero. Test cases SC8-T2 and SC8-T3 confirm validness of the change.

Implemented fix resolves the issue.



# [M-11] Possible role exploit using reentrancy vector in AppNFT

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol#L73-L92>

**Description:**

`AppNFT.safeTransferFrom()` removes all roles of current owner and transfers NFT to a different user. The problem is that `super.safeTransferFrom()` is performed before removing roles. This function additionally calls `onERC721Received()` for a magic value in case that recipient is smart contract, which may lead to a reentrancy attack, when there is a new owner already and all the roles are not cleared yet:

```
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId,
    bytes calldata data
) public
    override
{
    super.safeTransferFrom(from, to, tokenId, data);
    _removeAllRoles(tokenId); // @audit roles are removed after transfer. Reentrancy attack may
}
```

**Recommended Mitigation Steps:**

Please switch lines to make sure that all current user roles are removed before transferring the AppNFT.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.





# [M-12] Gas griefing and DoS app resources change

**Current Status:**  
**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol#L643-L696>

**Description:**

AppNFTs can be freely traded and transferred to others. This also transfers ownership of currently created apps. This can be exploited by a malicious actor, who can inflate app resources arrays to a big number, making all app updates very costly, or even impossible to perform. The root cause is that `nftSubnetResource` length once set, cannot be diminished:

```
function setParamSubnetResource(
    uint256 nftID,
    uint256 appID,
    uint256 subnetID,
    uint16[] memory curResource,
    uint16[] memory newResource,
    uint8[] memory newMul
)
internal
{
    uint256 subLen = nftSubnetResource[nftID][subnetID].length;
    uint8[] memory oldMul = appCurrentReplica[nftID][appID][subnetID];
    ....
    uint256 maxLen = max(subLen, newResource.length); // @audit when nftSubnetResource is set once to a
really big value, subnetResource array size cannot be diminished later

    uint32[] memory subnetResource = new uint32[](maxLen);
    ...
    nftSubnetResource[nftID][subnetID] = subnetResource;
```

Additionally, after London hard fork, adding support for [EIP-1559](#), block size is not constant 15M gas, but can be increased based up to 30M based on demand.



## [M-12] Gas griefing and DoS app resources change

This opens additional attack vector, where malicious app owner may update `resourceArray`, `nftSubnetResource` and `newMul` in a way, that that will require over 15M gas to process. This way it will be only possible to update the app if the block space demand is exceptionally high, soft DoSing the update.

### Recommended Mitigation Steps:

Enforce hard cap on maximum resource array. Given that the types of resources will be rather constant: CPU, memory, disc, networking, etc., setting it to max 10 or 20 seems to be sufficient mitigation, while not compromising protocol designed functionalities.

### Mitigation:

The logic is changed here.

Explanation:

We have resource categories, and in each category, we have packages.

CPU-MEMORY INSTANCE TYPES

C1, C2, C3, C4 .. C10

Eg:- CPU Standard, CPU Intensive, Nvidia T100 GPU etc

STORAGE INSTANCE TYPES:

S1, S2, S3, S4, S5

Eg: hard disk, solid state disk

BANDWIDTH types:

B1, B2, B3

Earlier logic was to have all of them in a single array, where each index in the array corresponds to a package.

[C1, C2, C3, S1,B1, C4, S2 ....]



## [M-12] Gas griefing and DoS app resources change

Newer logic is to save all the packages with resource type ID:

C1 - 0

C2 - 1

S1 - 2

B1 - 3

C3 - 4

...

So when we call `createApp`, we should pass resource count and the corresponding resource types:

eg: We want 3 instances of C3, 2 instances of S1, and 1 instance of B2.

We would have to pass the below arrays:

`resourceType: [4, 2, 3]`

`resourceCount:[3,2,1].`

New design resolves the issue with resources overflow.



# [M-13] The same app may be deleted multiple times, breaking protocol invariants

**Current Status:**  
**Fixed**

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol#L245-L283>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol#L1053-L1068>

## Description:

`ContractBasedDeployment.deleteApp()` is used to remove an app from a subnet. However, it does not check if the app was not already deleted. It may lead to many unexpected situations for off-chain clients and breaks following invariants:

- 256 max apps per subnet - deleting the same app multiple times may allow user to register over 256 apps
- deleting a subnet with single app removes a subnet. By constantly removing the same app, user may remove a subnet, having active apps inside.
- This allows subnet and app manipulation, e.g. function `getAppList()`, `updateCID()` will still work, even when driving app count to 0, which removes subnet.

## Recommended Mitigation Steps:

1. Check if app active Status: is `true` before deleting it
2. remove app from `appSubnetBitmap[nftID][appID]` bitmap when deleting it
3. Add appropriate tests for this case



## **[M-13] The same app may be deleted multiple times, breaking protocol invariants**

### **Mitigation:**

Team resolved the issue as recommended and tested it's validness creating test case SD53.

Implemented fix resolves the issue.



# **[M-14] Withdrawing misbehaving operator NFT does not stop the revenue generation**

**Current Status:**  
**Acknowledged**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L662-L674>

**Description:**

StackOS admin (GlobalDAO) has a possibility to punish a misbehaving subnet or cluster, by taking over DarkMatterNFT associated with this account.

1. This action does not automatically pause a subnet/cluster. Users have to pay revenue for the time that the operator is possibly not functioning.
2. In case of admin account compromise, a malicious actor can withdraw all DarkMatterNFTs from the smart contract.

**Recommended Mitigation Steps:**

Please consider stopping a malicious party, making them not accrue revenue anymore. Concerning centralization risk, please consider changing logic from withdrawing the NFT of a misbehaving party to blacklisting it in the system, and allowing its owner to withdraw it. This disincentivizes potential bad actors from stealing the NFTs.



## [M-14] Withdrawing misbehaving operator NFT does not stop the revenue generation

### Mitigation:

A malicious cluster can be stopped by decreasing its weight, deducting its staked token, or by delisting it which will stop the revenue distribution. A delisted subnet means no more clusters can join, and existing clusters can withdraw their NFT and staked tokens. Word of mouth and customer reviews can also discourage users from using misbehaving subnets. We are still deciding about stopping the revenue from flowing to a delisted subnet.

Team decided not to resolve the issue on-chain, but considers automating the action using on-chain code. We advise creating a robust off-chain agent to notify the team about such events and allowing them to react in timely manner.



# [M-15] Delisting a subnet does not stop revenue streaming for users using it

**Current Status:**  
**Acknowledged**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L428-L478>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalanceCalculator.sol#L442-L478>

**Description:**

In case that a subnet is delisted, there is no automatic mean to stop charging its users. This mean that until a user does not update their usage properties, they will be charged for every second that the subnet was not operating. This is hurtful for them and may be exploited by a malicious actors, to generate revenue for not operating the subnet.

**Recommended Mitigation Steps:**

Keep track of the time when subnet gets delisted and charge users up to the point when subnet was operating.

**Mitigation:**

Currently, the user can disable their app from running in that subnet, by setting the multiplier values for that subnet to zero, which will stop revenue from flowing to that subnet.

Team decided not to resolve the issue on-chain. We advise creating a robust off-chain agent to notify users about such events and allowing them to react in timely manner.





# [M-16] Not using safe operations may make XCTMinter unfunctional after stablecoin address change

Current Status:  
Fixed

Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L101>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L110>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L80-L86>

Description:

XCTMinter uses a stablecoin (currently USDC) to mint a pegged XCT. However it can be changed by admin to any other ERC20:

```
function changeUSDCAddress(IERC20Upgradeable _StableCoinToken)
external
onlyRole(DEFAULT_ADMIN_ROLE)
{
    emit ChangedStableCoinToken(StableCoinToken, _StableCoinToken);
    StableCoinToken = _StableCoinToken;
}
```

In case it is done, a special care has to be taken, given that many ERC20, like USDT, have uncommon behavior, e.g. reverting on non-zero to non-zero approval, not returning boolean at all, that may break the contract, making XCT tokens not exchangeable for stablecoin.



# [M-16] Not using safe operations may make XCTMinter unfunctional after stablecoin address change

## Recommended Mitigation Steps:

Use SafeTransferLib for all IERC20 related operations concerning transfer. Use `forceApprove()` or `safeIncreaseAllowance()` to account for tokens that revert on setting non-zero to non-zero approval.

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [M-17] Admin can make XCTMinter disfunctional by changing or withdrawing stablecoin

**Current Status:**  
**Fixed**

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L80-L86>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L97-L104>

## Description:

**XCTMinter** admin has possibility to change stablecoin address or withdraw it. There are three main risks concerning that approach:

1. New stablecoin address is not guaranteed to have the same number of decimals. GUSD has 2 decimal places, while USDC and USDT have 6 decimals on all chains but BSC, where they have 18 decimal places.
2. New stablecoin will not be escrowed in the smart contract. Anyone trying to exchange XCT for a stablecoin will fail.
3. In case of a key compromise, a malicious actor can change the address to a malicious or worthless ERC20, or just use **withdrawUSDC()** to steal all escrowed stablecoins.

## Recommended Mitigation Steps:

Change **stepUpFactor** together with stablecoin address change. Require that the new **stepUpFactor** is not 0.

Concerning centralization risk, please make sure that the admin account is at least a multisig with a delayed execution (timelocked function). Ideally, make **changeUSDCAddress()** and **withdrawUSDC()** two step operation, requiring few days delay and emit an event on the start of the action.



## **[M-17] Admin can make XCTMinter disfunctional by changing or withdrawing stablecoin**

### **Mitigation:**

Now when stable coin address is changed, the stepUpFactor needs to be set, and a certain amount of new stable coin tokens needs to be staked into the contract.

Implemented fix resolves the issue. It should be noted that there is still some amount of trust is required, because compromised admin can stake different amount of tokens than previous stablecoin, or stake a custom ERC20 that does not have a market value.



# [M-18] Missing tests

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L1-L341>

**Description:**

There are many places throughout the codebase, which do not have any tests to cover the functionality. It's very important to have 100% code coverage, which is standard in smart contracts codebases. It allows to eliminate bugs early and verify if the code works in intended ways. One of the contracts is XCTMinter, which is crucial, because it gets stablecoins from users and mints XCT tokens. It has no coverage at all.

**Recommended Mitigation Steps:**

Use [solidity-coverage](#) npm package to verify the code coverage and add missing tests.

**Mitigation:**

Team resolved the issue as recommended.

The team added solidity-coverage and introduced tests for issues pointed in the audit.



# [M-19] Clusters are in risk of not getting paid and the funds being stuck in SubnetDAODistributor when setting revenue is performed often

**Current Status:**

Fixed

**Context:**

<https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L200-L214>

**Description:**

When calling `SubnetDAODistributor.assignRevenues()`, total revenues are distributed to clusters, based on their weights. There is no check on when last time it was called, so can anyone can call it in each block. In case that the amount of clusters is high, calculations in `uint256 rev = weight.mul(subRev).div(totalWeight)`; may lead to precision loss and receiving no funds. Given exemplary calculations:

`totalWeight = 1000000`

`clusterWeight = 10`

`subRev = 99999`

`rev = (10 * 99999) / 1000000 = 0.9999` , which gives 0 due to rounding in Solidity

Malicious actor can effectively lead to all clusters losing all the funds and locking it in the contract.



# **[M-19] Clusters are in risk of not getting paid and the funds being stuck in SubnetDAODistributor when setting revenue is performed often**

## **Recommended Mitigation Steps:**

Add time threshold, minimum amount threshold, or both, to make sure that a malicious actor cannot invoke this function grieving clusters.

## **Mitigation:**

After doing the calculations, this seems insignificant.

Consider 1 second per block

each block, we accumulate dust of  $-10^{(-18)}$  dollars

To reach 1 dollar, we need 31,709,791,983.7 years

Team acknowledges this issue, pointing that it has very low economic impact in practice.



# [L-1] `_safeMint` creates reentrancy attack possibility

## Current Status:

Fixed

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol#L58>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/DarkMatterNFT.sol#L58>

## Description:

When minting NFTs, the code uses `_safeMint` function of the OZ reference implementation. This function is safe because it checks whether the receiver can receive ERC721 tokens. This can prevent the case that an NFT will be minted to a contract that cannot handle ERC721 tokens. However, this external function call creates a security loophole. Specifically, the attacker can perform a reentrant call inside the `onERC721Received` callback. More detailed information on why reentrancy can occur - <https://blocksecteam.medium.com/when-safemint-becomes-unsafe-lessons-from-the-hypebears-security-incident-2965209bda2a>.

## Recommended Mitigation Steps:

Add OpenZeppelin `nonReentrant` modifier

## Mitigation:

The `nonReentrant` modifier is not added, as only a trusted minter address has permissions to mint appNFTs.

Team acknowledges the issue, pointing that only a trusted party will be able to invoke this function.





# [L-2] Wrong time check

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L375>

**Description:**

The timestamp expiry of the credit with the specific id is compared with the block.timestamp is wrong way.

**Recommended Mitigation Steps:**

```
-- creditsExpiry[sender][nftID].expiryTimestamp < block.timestamp,  
++ creditsExpiry[sender][nftID].expiryTimestamp <= block.timestamp,
```

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-3] Not blocking initialization in constructor leads to unexpected initialization of logic contract by anyone

Current Status:

Fixed

Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTERC20.sol#L14-L19>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L70-L82>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L55-L62>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterA.sol#L201-L209>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L18>

Description:


Leaving default constructor in logic contract of the proxy pattern should be avoided. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. While we were not able to identify any serious issues with current implementation, any future updates may introduce this attack vector and it should be fixed. This issue concerns all upgradeable contracts in scope.



# [L-3] Not blocking initialization in constructor leads to unexpected initialization of logic contract by anyone

## Recommended Mitigation Steps:

OpenZeppelin's official recommendation is to disable initializers in constructor:



```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-4] Sending data with ERC721 and ERC1155 token transfers may lead to unexpected results

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TokensRecoverableOwner.sol#L17-L19>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TokensRecoverable.sol#L17-L19>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/LinkNFTs.sol#L59-L64>

**Description:**

ERC1155 and ERC721 standards allow for sending bytes together with transfer, and some implementations may react in unexpected ways when the data is passed. In this case, a constant string "0x" is passed, which sends byte array with two characters inside.

**Recommended Mitigation Steps:**

Pass empty string in data parameter of ERC721 and ERC1155 transfer - ""

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-5] Wrong storage gap value may break storage layout in new contract version

## Current Status:

Fixed

## Context:

- <https://github.com/stacksofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/MultiAccessControlUpgradeable.sol#L344>

## Description:

`MultiAccessControlUpgradeable` implements storage gap, which prevents storage collisions in new versions. However, they are meant to be `50 - storage-slots-reserved` by standard. Only complying to it saves from storage collisions, otherwise they are still possible after upgrade. In case of this contract, there are 4 slots reserved for private mappings, and the `__gap` is 49 elements long.

## Recommended Mitigation Steps:

Use `50 - 4 = 46` value for storage gap, as there are 4 slots reserved for storage variables already

It's also recommended to use [OpenZeppelin's Upgrade Plugin](#)

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-6] Checks effects interactions pattern not followed

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L288-L294>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalance.sol#L593-L601>

**Description:**

In `SubscriptionBalance._withdrawBalance()` and `receiveRevenueForAddress()`, CEI pattern is not followed and the `XCT` token is sent before user balance is updated. While it does not pose a security risk at the moment, if not fixed, may be problematic in the future.

**Recommended Mitigation Steps:**

Update user balances first, and then sent `XCT` to the recipient.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-7] No check for ERC20 transfer return value

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L362-L366>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L587>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L621>

**Description:**

ERC20 `transfer()` and `transferFrom()` functions do not revert, but return `false` on failure according to the standard. While STACK and XCT tokens used throughout the codebase revert on failure, any future changes may accidentally be changed. This could lead to severe consequences.

**Recommended Mitigation Steps:**

Use OpenZeppelin's `SafeTransferLib` - like in `TokensRecoverable` contract - for dealing with ERC20 token transfers. Please also consider if it's beneficial to also add support for fee-on-transfer tokens.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-8] Subnet may have more active clusters than maxClusters

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L489-L492>

**Description:**

It's possible to list more clusters than `maxClusters` in `SubnetClusterB.approveListingCluster()`. That's because available spots check is joined cluster listing Status:

```
subnetClusters[subnetId][clusterId].listed != 3 ||  
totalClusterSpotsAvailable(subnetId) > 0
```

This means, that in case that cluster Status: is 1, first expression will evaluate to true and the check will be effective, allowing to go past `maxClusters`, which breaks the active cluster size invariant.

**Recommended Mitigation Steps:**

Change line 490 from

```
subnetClusters[subnetId][clusterId].listed != 3 ||  
totalClusterSpotsAvailable(subnetId) > 0
```

to

```
subnetClusters[subnetId][clusterId].listed != 3 &&  
totalClusterSpotsAvailable(subnetId) > 0
```

**Mitigation:**

Although the code has been changed, the above logic is valid, as clusters with listed value "1" are considered to be taking up a spot. In the newer version of the code, only clusters with status 3 are considered to be taking up a spot.

Implemented fix resolves the issue.





# [L-9] Replicating the same storage layout for diamond proxy may break the protocol after updating

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterA.sol#L22-L42>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L22-L42>

**Description:**

Diamond proxy requires nonstandard storage layout, due to the fact that each facet may be a different smart contract, and hence override the same storage slots. Please refer to the [EIP-2535 storage](#) for more info and rationale. [SubnetClusterA](#) and [SubnetClusterB](#) are part of a single diamond. The contract had to be split, because it exceeds 24 kB in size. Both of them share common storage, hence a special care has to be taken to not have different storage layout between both of the contracts. In case that any of them change storage layout, it may have disastrous consequences to the project.

**Recommended Mitigation Steps:**

1. Use a single abstract smart contract that both [SubnetClusterA](#) and [SubnetClusterB](#) will inherit from, keeping all the common code like constants, structs et al, there.
2. Split storage to be used by either one or another smart contract. Strive for each smart contract storage independence, by gathering functions using the same storage in one smart contract
3. Use unstructured storage, diamond storage or app storage, to store variables in unstructured matter, to make sure that there is no storage collision



## **[L-9] Replicating the same storage layout for diamond proxy may break the protocol after updating**

### **Mitigation:**

A single abstract smart contract that contains data is used for both SubnetClusterA and SubnetClusterB.

Implemented fix resolves the issue. Special care has to be taken when updating the diamond, because storage layout mismatch between new and old contracts versions could still lead to serious issues.



# [L-10] Implemented cluster signup logic does not conform to requirements

Current Status:

Fixed

Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L321-L411>

Description:

Confirmed behavior should be: if a subnet is public, then anyone can create a cluster for it. If a subnet is private, then only whitelisted accounts can join. However in `SubnetClusterB.clusterSignUp()`, this operation is implemented differently:

```
    {
        bool isWhitelisted = false;
        for (uint256 i = 0; i < whiteListedClusters[subnetId].length; i++)
            if (whiteListedClusters[subnetId][i] == ownerAddress) {
                subnetClusters[subnetId][clusterId].listed = 2; // whitelisted clusters are approved as
they signup
                SubnetDAODistributor.setClusterWeight(
                    subnetId,
                    clusterId,
                    DefaultWhitelistedClusterWeight
                );
                isWhitelisted = true;
                break;
            }

        if(!isWhitelisted)
        {
            subnetClusters[subnetId][clusterId].listed = 1;
        }
    }
```

In the code above:

1. Anyone technically will be eligible to join. If their address is not whitelisted, their `listed` Status: will be set to 1.
2. In case of a public subnet, where anyone can join, `listed` Status: will be set to 1 and the user has to be whitelisted either way.



## [L-10] Implemented cluster signup logic does not conform to requirements

Additionally, there is no way of listing a cluster for public subnets, because of this check:

```
function addClusterToWhitelisted(
    uint256 subnetId,
    address[] memory _whitelistAddresses
) external {
    _requireNotPaused();
    require(
        (subnetAttributes[subnetId].subnetType & 1) > 0,
        "Already public subnet"
    );
}
```

That means that public clusters have to be manually approved via `SubnetClusterB.approveListingCluster()`, which is opposite of how it should work.

### Recommended Mitigation Steps:

Change the code to match expected logic by:

- failing fast in case that the account is not whitelisted, to prevent locking DarkMatterNFT in smart contract that the user is not supposed to
- automatically list cluster for public subnets

### Mitigation:

The code was updated. Anyone can join a public subnet, but will have their listed state as waiting (listed = 1). The subnetDAO should set its state to active (listed = 2)

For a private subnet, the clusters joining have to be whitelisted, or else the transaction fails.

Implemented fix resolves the issue.



# [L-11] Gas griefing and DoS when subnet adds too much of whitelisted addresses

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterB.sol#L374-L390>

**Description:**

When a new cluster is added, in worst case scenario all `whiteListedClusters[subnetId]` array elements are iterated over to check if an address is whitelisted:

```
    {
        bool isWhitelisted = false;
        for (uint256 i = 0; i < whiteListedClusters[subnetId].length; i++)
            if (whiteListedClusters[subnetId][i] == ownerAddress) {
                subnetClusters[subnetId][clusterId].listed = 2; // whitelisted clusters are approved as
they signup
                SubnetDAODistributor.setClusterWeight(
                    subnetId,
                    clusterId,
                    DefaultWhitelistedClusterWeight
                );
                isWhitelisted = true;
                break;
            }

            if(!isWhitelisted)
            {
                subnetClusters[subnetId][clusterId].listed = 1;
            }
    }
```

may grow to a point when it either cost substantial amount of gas, or even DoSes registering new clusters.



# [L-11] Gas griefing and DoS when subnet adds too much of whitelisted addresses

## **Recommended Mitigation Steps:**

Use reverse mapping holding whitelisted addresses, to bypass unbounded loop iteration, or limit max amount of whitelisted accounts that a subnet may add. Generally, try to avoid unbounded loops as much as possible.

## **Mitigation:**

Team resolved the issue by changing array to a mapping: `mapping(uint256 => uint256) public totalClustersSigned`.

Implemented fix resolves the issue.



# [L-12] Missing events emits in important functions

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/Subscription.sol#L338-L352>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/Subscription.sol#L313-L336>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/Subscription.sol#L299-L311>

**Description:**

Events provide valuable information to the users, and are a standard way to communicate changes to the offchain clients like TheGraph. Hence, it's important that any significant change emits an event with as much indexed parameters as possible, in order to ease data scrapping and allow alerting tools a way to warn users in case of a problem. We identified following functions that don't emit events for important actions:

- `Subscription.setSupportFactorForNFT()`
- `Subscription.addPlatformAddress()`
- `Subscription.addSupportAddress()`

**Recommended Mitigation Steps:**

Add missing events to the listed functions. Make sure that ALL events are using indexed parameters as much as possible

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-13] Misleading use of `_safeMint()` in `AppNFT.mint()` may lead to misuse of the NFT

**Current Status:**

Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol#L54-L60>

**Description:**

According to "*principle of least astonishment*" described in The Dangers of Surprising Code, the function should do exactly what it's supposed to by its name. In case of `AppNFT.mint()`, it uses internal `_safeMint()`, which is not expected, and may lead to misuse of the NFT. This might be a real problem, given that the AppNFT is supposed to be used by app creators on StackOS platform.

**Recommended Mitigation Steps:**

Consider using internal `_mint()` function in `mint()`, and create separate `safeMint()`, which calls internal `_safeMint()` and let app owners decide which function they want to use. Make sure that both of them can be called only by a whitelisted `msg.sender`.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.





# [L-14] Wrong calculations of app subnets in `ContractBasedDeployment.getSubnetsOfApp()`

**Current Status:**  
Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol#L1141-L1148>

**Description:**

`ContractBasedDeployment.getSubnetsOfApp()` is meant to return an array of subnets active for an app. Internally, it goes over a list of subnets for an NFT twice - first time to count list of active subnets, second time to fill the return array only with active subnets. The problem is a bitwise operation used in second loop: `(activeBitmap & i) > 0`. It's incorrect, as it will match any bit on the same place in `activeBitmap` and `i`, for example `0b01` will be matched every odd number:

```
for(uint i = 0; i < subnetLen; i++)
{
    if((activeBitmap & i) > 0)
    {
        subnetList[j] = nftAllSubnets[nftID][i];
        j++;
    }
}
```

This will return falsified results. Even though the function is not used on-chain, it should be corrected, as it may impact off-chain code.



# [L-14] Wrong calculations of app subnets in **ContractBasedDeployment.getSubnetsOf App()**

## **Recommended Mitigation Steps:**

Change bitwise operation to:

`(activeBitmap & (1 << i)) > 0`

in line 1141.

## **Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [L-15] Allowing arbitrary UniswapV2Router address may lead to user funds being lost

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L244>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L185>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L164>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/XCTMinter.sol#L131>

**Description:**

When performing swaps in **XCTMinter**, user can pass any arbitrary **UniswapV2Router** address. Even though the contract is protected by **nonReentrant** flags disallowing to skew the accounting in the middle of the transaction, the very fact that even malicious router can be passed may be exploited. While user will only be credited to the stablecoin difference after swap, a malicious actor can prepare a malicious page using valid StackOS contracts to lure users and substitute just the input parameter for a malicious router, stealing the funds from users.

**Recommended Mitigation Steps:**

In case that many Uniswap forks are to be supported per chain, please whitelist all router contracts and enforce only those are accepted. If only one is meant to be supported, set it in the storage.



# **[L-15] Allowing arbitrary UniswapV2Router address may lead to user funds being lost**

## **Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [I-1] Change custom pausable design pattern implementation

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/ContractBasedDeployment.sol#L20>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubnetClusterA.sol#L18>

**Description:**

There is an inconsistency across the codebase related to how the Pausable design pattern is implemented. It's advisable to prefer battle-tested code over reimplementing common patterns. In some contracts like Subscription for example inherit from PausableUpgradeable, which is totally correct.

**Recommended Mitigation Steps:**

Replace the `_paused` modifier in `ContractBasedDeployment` with the one from Pausable from OpenZeppelin, since it is well-tested, optimized, and secure.

**Mitigation:**

Team kept custom pausing logic, due to the fact that it uses external contract to check for pauser role.

Team left current logic as-is.



# [I-2] Use a more recent version of solidity consistently across the whole codebase

**Current Status:**

**Fixed**

**Context:**

- All contracts

**Description:**

Currently, different versions are used (0.8.2, ^0.8.8). Using a floating pragma ^0.8.8 statement is discouraged as code can compile to different bytecodes with different compiler versions. Use a stable pragma statement to get a deterministic bytecode.

**Recommended Mitigation Steps:**

We recommended to use 0.8.19 Solidity version in all contracts to get all compiler features, bug fixes, and optimizations. It is the last available version that does not have breaking changes (new PUSH0 opcode), which is not yet supported on all EVM compatible chains.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix (using v0.8.19) resolves the issue.



# [I-3] Usage of SafeMath is deprecated in Solidity >=v0.8.0

**Current Status:**  
Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/SubscriptionBalanceCalculator.sol#L16>

**Description:**

Since Solidity v0.8.0, all math operations revert on underflow and overflow. Usage of SafeMath is then deprecated and incurs additional gas footprint.

**Recommended Mitigation Steps:**

Remove all instances of `SafeMath`

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [I-4] Duplicated Counters library definition

**Current Status:**  
**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/AppNFT.sol#L9-L31>
- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/DarkMatterNFT.sol#L9-L31>

**Description:**

**Counters** library is defined in both **DarkMatterNFT** and AppNFT. It's implementation is the same in both contract, hence it is advised to move it into a separate file and reference them in the contracts. It would lessen the mental overhead for the next developers and auditors. Additionally, it protects from accidental logic change in one of the implementations.

**Recommended Mitigation Steps:**

Move **Counters** library to separate file and reference it in both NFT implementations.

**Mitigation:**

Team resolved the issue as recommended.

Implemented fix resolves the issue.





# [I-5] Unnecessary address casting

## Current Status:

Fixed

## Context:

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/DarkMatterNFT.sol#L61>

## Description:

Performing the MinterAddress casting is unnecessary since it's stored as an address and can be newly set only by the owner again with an address.

## Recommended Mitigation Steps:

Remove the casting and just compare both addresses to ensure the right access control.

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [I-6] Remove commented functionalities

**Current Status:**

Fixed

**Context:**

- All Contracts

**Description:**

In most of the contracts, there are many commented methods and variables.

**Recommended Mitigation Steps:**

It is generally good practice to remove or properly manage unused or obsolete code to maintain clarity and minimize confusion for other developers who may interact with the contract in the future.

**Mitigation:**

Team removed all dead code.

Implemented fix resolves the issue.



# [I-7] All contracts are unlicensed

## Current Status:

Fixed

## Context:

- All Contracts

## Description:

All of the contracts in the scope are unlicensed. Without a license, the default copyright laws apply, which can restrict the usage, distribution, and modification of the contract. This lack of clarity can deter potential users, contributors, or collaborators who may be hesitant to engage with the contract due to legal uncertainties. Additionally, many tools rely on open-source licenses. By using an appropriate license, you ensure that your contract can be compatible with these tools.

## Recommended Mitigation Steps:

Add an appropriate SPDX license.

## Mitigation:

Team resolved the issue as recommended.

Implemented fix resolves the issue.



# [I-8] Missing NatSpec

**Current Status:**

**Fixed**

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts>

**Description:**

None of the smart contract (apart from 3rd party ones) uses NatSpec. It is considered bad practice and makes the code harder to read and reason about. Having NatSpec, apart from easier code comprehension, builds credibility of the protocol, because all mainstream blockchain explorers are able to display NatSpec to the users.

**Recommended Mitigation Steps:**

Add NatSpec to all contracts and functions definitions.

**Mitigation:**

Team promised to add missing NatSpec.

At the time of finishing the report, the team told that they will add the NatSpec. It was not verified, however it does not have impact on the protocol code.



# [I-9] Unclear TestERC20/STACK implementation

**Current Status:**  
Fixed

**Context:**

- <https://github.com/stackosofficial/stackos-v2-contracts/blob/df19c89ec61075ec47ed22ec0cce726683aef329/contracts/TestERC20.sol#L9>

**Description:**

The owner has all the tokens in their possession. Additionally, TestERC20 is actually used as a STACK token which is not described in any way. Both of these lead to serious confusion.

**Recommended Mitigation Steps:**

Redesign the logic and its implementation.

**Mitigation:**

Team promised to add missing NatSpec.

At the time of finishing the report, the team told that they will add the NatSpec. It was not verified, however it does not have impact on the protocol code.



# Recommendations

## [R-1] Centralization risks around GlobalDAO

### Description

**GlobalDAO**, which is an admin account, has many privileges, like managing subnets and clusters, removing NFTs, withdrawing stablecoins from XCTMinter, changing all crucial system settings. While it is required for the system to function properly and punish misbehaving parties, a special care has to be taken, to minimize potential protocol damage in case of account compromise.

We recommend ensuring that GlobalDAO/admin account is OpenZeppelin's Timelock, with at least two independent parties required to sign an admin action via a multisig account. In the future, when the project decides to decentralize, consider using Compound's GovernorBravo and Timelock, where token holders will have power to change the protocol.

## [R-2] Misleading Registration / SubnetCluster naming

### Description

**Registration** contract was replaced with **SubnetCluster**, which due to contract size constraints had to be split into two separate smart contracts, connected by diamond pattern proxy. However, all the code references still point to **Registration** contract.

We recommend creating proper interfaces for **ISubnetClusterA**, **ISubnetClusterB**. Additionally create **ISubnetCluster** interface, inheriting from two previous contracts and rename all **Registration** references to **ISubnetCluster**.



# Recommendations

**[R-3] Bridge role granted despite the project not being fully multichain yet**

## Description

Bridge role allows withdrawing all owner balance, managing fees and creating subscriptions. It's meant to be a part of future multichain infrastructure and have additional safeguards on sending chain side. However, [GlobalDAO](#) is granted bridge role, which in case of account compromise could be a serious problem.

We recommend not granting [BRIDGE\\_ROLE](#) to anyone, even [GlobalDAO](#) until multichain support is added and audited.

