OBELISK

Part of Tibereum Group

AUDITING REPORT

Version Notes

Version	No. Pages	Date	Revised By	Notes
1.0	Total: 51	2023-10-11	Donut, DoD4uFN	Audit Final

Audit Notes

Audit Date	2023-03-13 - 2023-10-10
Auditor/Auditors	DoD4uFN, Plemonade
Auditor/Auditors Contact Information	contact@obeliskauditing.com
Notes	Specified code and contracts are audited for security flaws. UI/UX (website), logic, team, and tokenomics are not audited.
Audit Report Number	OB565854111

Disclaimer

This audit is not financial, investment, or any other kind of advice and is for informational purposes only. This report is not a substitute for doing your own research and due diligence. Obelisk is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Obelisk has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Obelisk is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. The audit is paid by the project but neither the auditors nor Obelisk has any other connection to the project and has no obligations other than to publish an objective report. Obelisk will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Obelisk assumes that the provided information and material were not altered, suppressed, or misleading. This report is published by Obelisk, and Obelisk has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Obelisk. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Obelisk Auditing

Defi is a relatively new concept but has seen exponential growth to a point where there is a multitude of new projects created every day. In a fast-paced world like this, there will also be an enormous amount of scams. The scams have become so elaborate that it’s hard for the common investor to trust a project, even though it could be legit. We saw a need for creating high-quality audits at a fast phase to keep up with the constantly expanding market. With the Obelisk stamp of approval, a legitimate project can easily grow its user base exponentially in a world where trust means everything. Obelisk Auditing consists of a group of security experts that specialize in security and structural operations, with previous work experience from among other things, PricewaterhouseCoopers. All our audits will always be conducted by at least two independent auditors for maximum security and professionalism.

As a comprehensive security firm, Obelisk provides all kinds of audits and project assistance.

Audit Information

The auditors always conduct a manual visual inspection of the code to find security flaws that automatic tests would not find. Comprehensive tests are also conducted in a specific test environment that utilizes exact copies of the published contract.

While conducting the audit, the Obelisk security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Obelisk assesses the risks and assigns a risk level to each section together with an explanatory comment. Take note that the comments from the project team are their opinion and not the opinion of Obelisk. This document is a summary of the findings that the auditors found during their audit and is no guarantee for how safe the contracts are.

Table of Contents

Project Information

Name	Strain
Description	DeFi Staked Crypto Cannabis Collectables.
Website	https://strainnft.com/
Contact	https://twitter.com/strainNFT
Contact information	@crispybambino on Discord
Token Name(s)	N/A
Token Short	N/A
Contract(s)	See Appendix A
Code Language	Solidity
Chain	Fantom

Audit of Strain

Obelisk was commissioned by Strain on the 8th of March 2023 to conduct a comprehensive audit of Strains’ contracts. The following audit was conducted between the 13th of March 2023 and the 10th of October 2022. Two of Obelisk's security experts went through the related contracts manually using industry standards to find if any vulnerabilities could be exploited either by the project team or users.

The reason for the long time lag between the start of the audit and the finish is that we were waiting on the team to finish tasks that needed to be done in order to complete the audit.

Overall, there were multiple findings with different levels of severity in the initial contracts. After the initial report was created, the Strain team worked on solving the majority of issues before we could finalize the audit. All but two issues of notice, issue #14 and issue #16 are solved. **Note that no on-chain analysis has been done on the contracts as they were not deployed at the time of this audit. We cannot confirm that the deployed contracts are the audited ones.**

The informational findings are good to know while interacting with the project but don’t directly damage the project in its current state, hence it’s up to the project team if they deem that it’s worth solving these issues, however, please take note of them.

The team has not reviewed the UI/UX, logic, team, or tokenomics of the Strain project**.**

This document is a summary of the findings that the auditors found during their audit. Please read the full document for a complete understanding of the audit.

Summary Table

Code Analysis

Finding	ID	Severity	Status
Unlimited Free RewardPerNFT Tokens By Creating 0 Strains	#0001	High Risk	Closed
Broken Approval Mechanism	#0002	High Risk	Closed
NFT Genes Can Be Manipulated	#0003	High Risk	Closed
Creating Multiple Strains Only Gives Out Reward For A Single One	#0004	Medium Risk	Closed
Owner Can Choose Arbitrary TokenURI	#0005	Medium Risk	Closed
Unbounded Loop	#0006	Medium Risk	Closed
Emergency Withdraw Checks Effects Interactions	#0007	Low Risk	Closed
Claiming Rewards Is Time Based	#0008	Low Risk	Closed
Transferring Zero Rewards	#0009	Low Risk	Closed
No Events Emitted For Changes To Protocol Values	#0010	Informational	Closed
Remove Unnecessary Code	#0011	Informational	Closed
Gas Savings	#0012	Informational	Open
Accumulated Shares Not Updated Correctly	#0013	Low Risk	Closed
TokenURI Can Be Modified By Whitelisted Admins	#0014	Low Risk	Open
NFTs Can Be Transferred Without Finalizing The Mint	#0015	Low Risk	Closed
Rewards Distribution Is Callable Only By The Admin Role	#0016	Medium Risk	Open

On-Chain Analysis

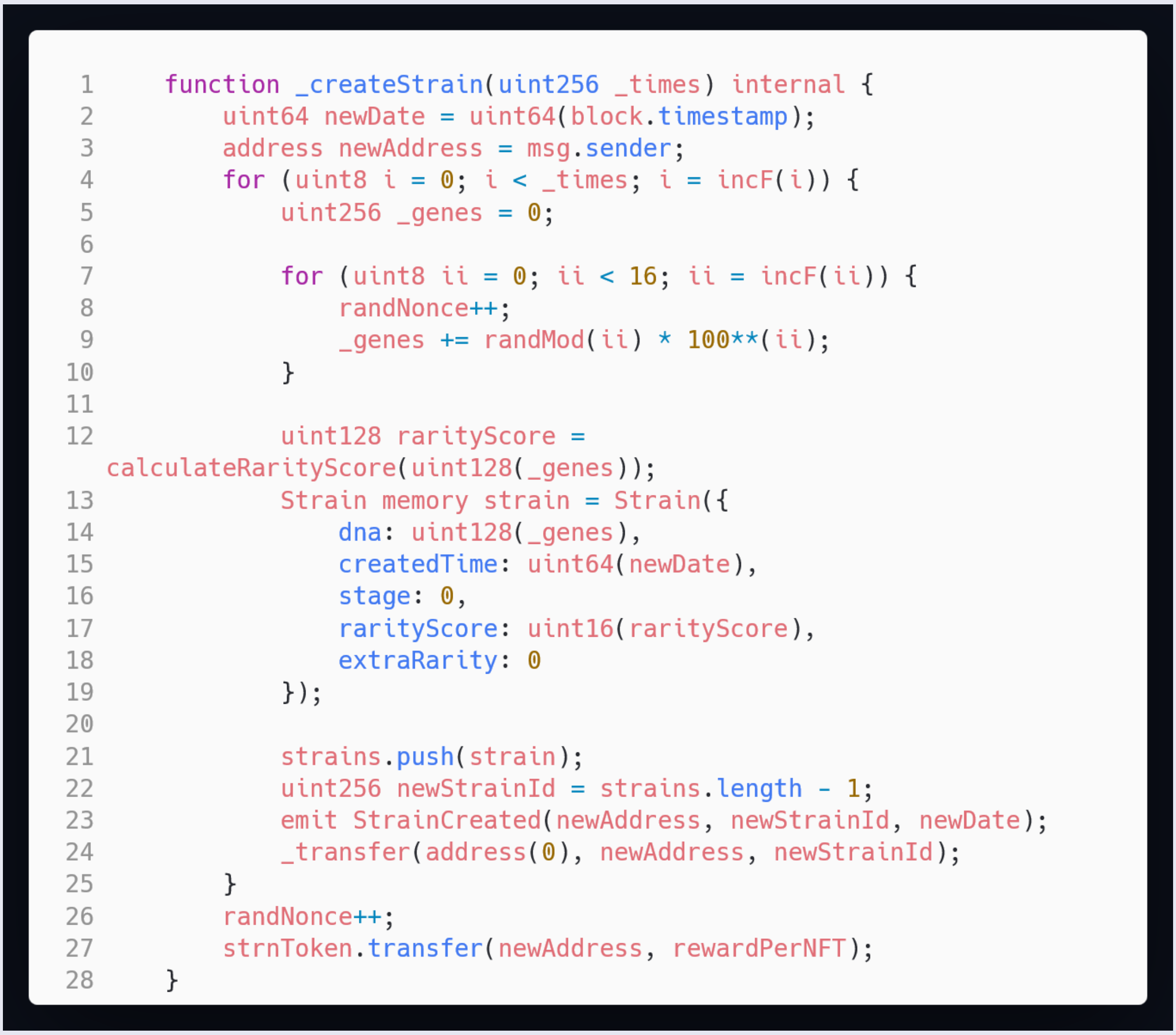
Finding	ID	Severity	Status
Not Analyzed	-	-	-

Findings

Code Analysis

Unlimited Free RewardPerNFT Tokens By Creating 0 Strains

FINDING ID	#0001
SEVERITY	High Risk
STATUS	Closed
LOCATION	StrainFactory.sol -> 130-157



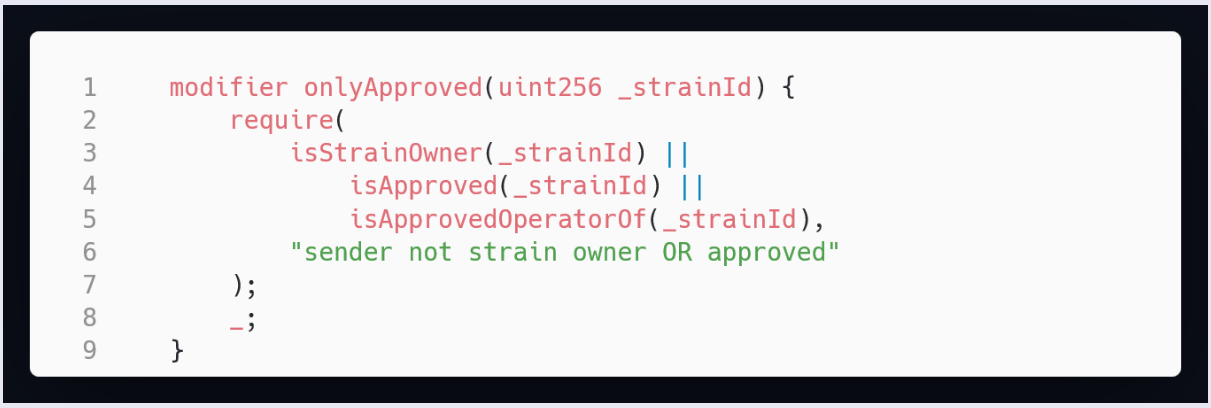
```
function _createStrain(uint256 _times) internal {
    uint64 newDate = uint64(block.timestamp);
    address newAddress = msg.sender;
    for (uint8 i = 0; i < _times; i = incF(i)) {
        uint256 _genes = 0;

        for (uint8 ii = 0; ii < 16; ii = incF(ii)) {
            randNonce++;
            _genes += randMod(ii) * 100**(ii);
        }

        uint128 rarityScore = calculateRarityScore(uint128(_genes));
        Strain memory strain = Strain({
            dna: uint128(_genes),
            createdTime: uint64(newDate),
            stage: 0,
            rarityScore: uint16(rarityScore),
            extraRarity: 0
        });

        strains.push(strain);
        uint256 newStrainId = strains.length - 1;
        emit StrainCreated(newAddress, newStrainId, newDate);
        _transfer(address(0), newAddress, newStrainId);
    }
    randNonce++;
    strnToken.transfer(newAddress, rewardPerNFT);
}
```

DESCRIPTION	When invoking _*createStrain* from *createStrain* with _*times* = 0, the user will not be charged and will receive *rewardPerNFT* quantity of strnTokens.
RECOMMENDATION	Calculate the *rewardPerNFT* variable based on the number of tokens minted and make sure that times are greater

	than zero so *randNonce* can't be manipulated.
RESOLUTION	The project team has implemented the recommended changes.

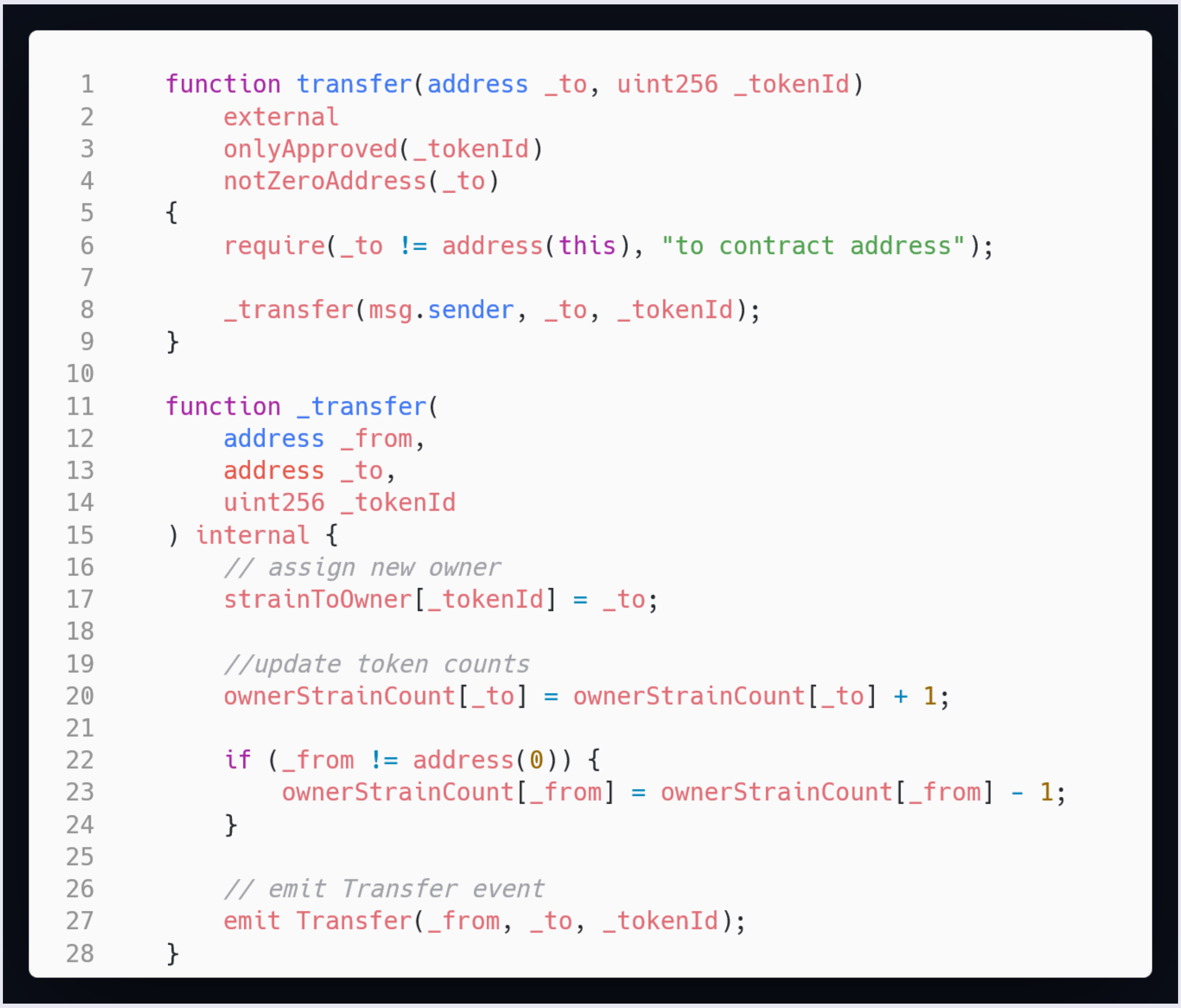
Broken Approval Mechanism

FINDING ID	#0002
SEVERITY	High Risk
STATUS	Closed
LOCATION	StrainContract.sol -> 70-78



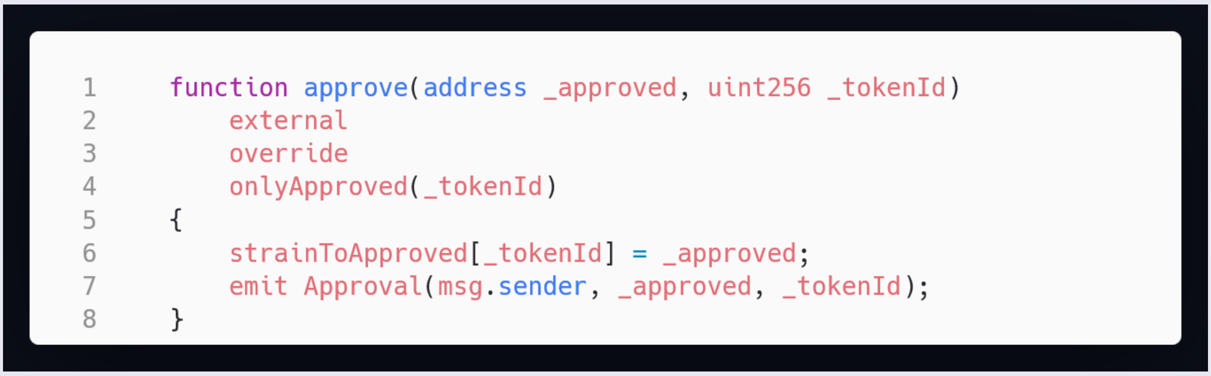
```
modifier onlyApproved(uint256 _strainId) {
    require(
        isStrainOwner(_strainId) ||
            isApproved(_strainId) ||
            isApprovedOperatorOf(_strainId),
        "sender not strain owner OR approved"
    );
    _;
}
```

LOCATION	StrainContract.sol -> 220-247



```
function transfer(address _to, uint256 _tokenId)
    external
    onlyApproved(_tokenId)
    notZeroAddress(_to)
{
    require(_to != address(this), "to contract address");

    _transfer(msg.sender, _to, _tokenId);
}

function _transfer(
    address _from,
    address _to,
    uint256 _tokenId
) internal {
    // assign new owner
    strainToOwner[_tokenId] = _to;

    //update token counts
    ownerStrainCount[_to] = ownerStrainCount[_to] + 1;

    if (_from != address(0)) {
        ownerStrainCount[_from] = ownerStrainCount[_from] - 1;
    }

    // emit Transfer event
    emit Transfer(_from, _to, _tokenId);
}
```

LOCATION	StrainContract.sol -> 255-262

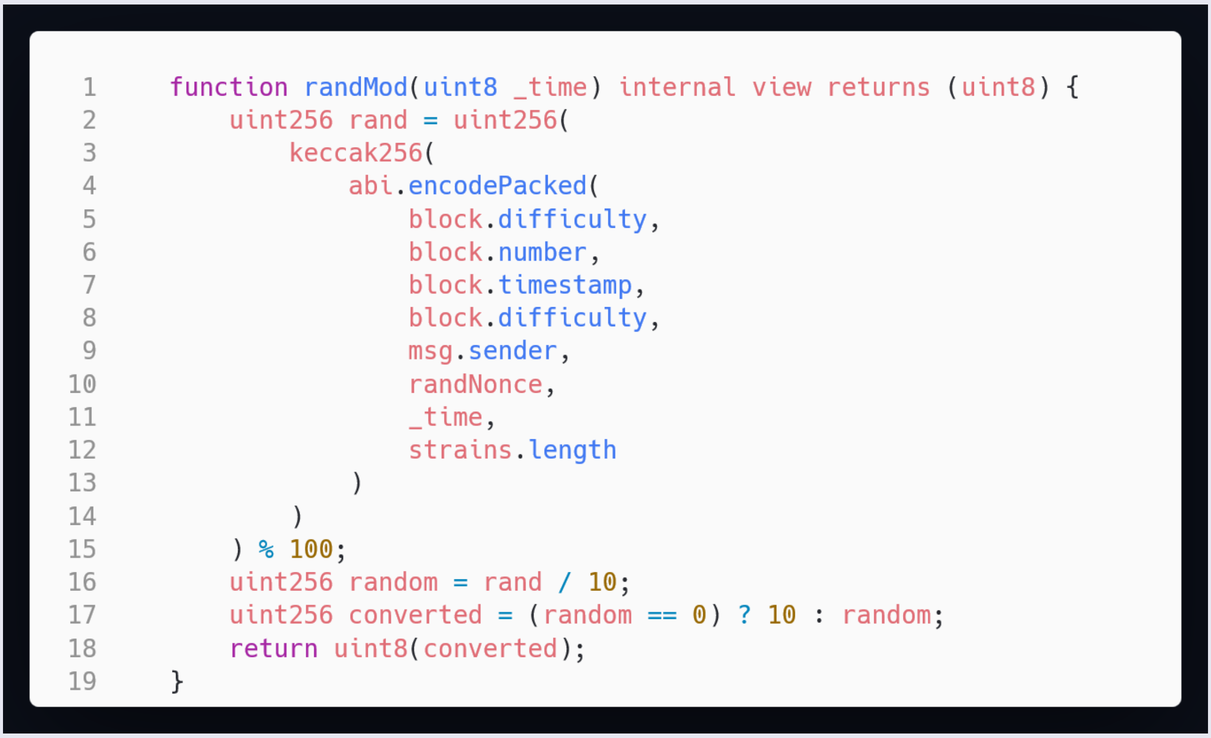


```
function approve(address _approved, uint256 _tokenId)
    external
    override
    onlyApproved(_tokenId)
{
    strainToApproved[_tokenId] = _approved;
    emit Approval(msg.sender, _approved, _tokenId);
}
```

DESCRIPTION	The _*transfer* function should clear the token approvals stored in *strainToApproved*. Currently, if a token is approved and then transferred, it can be stolen from the receiver because the approval isn't automatically cleared. This vulnerability could be exploited to sell an NFT while still receiving rewards for it.
RECOMMENDATION	The _*transfer* function should include a step that clears the approval for the transferred token, ensuring that it can't be stolen by the previous owner or any other unauthorized party.
RESOLUTION	The project team has implemented the recommended changes.

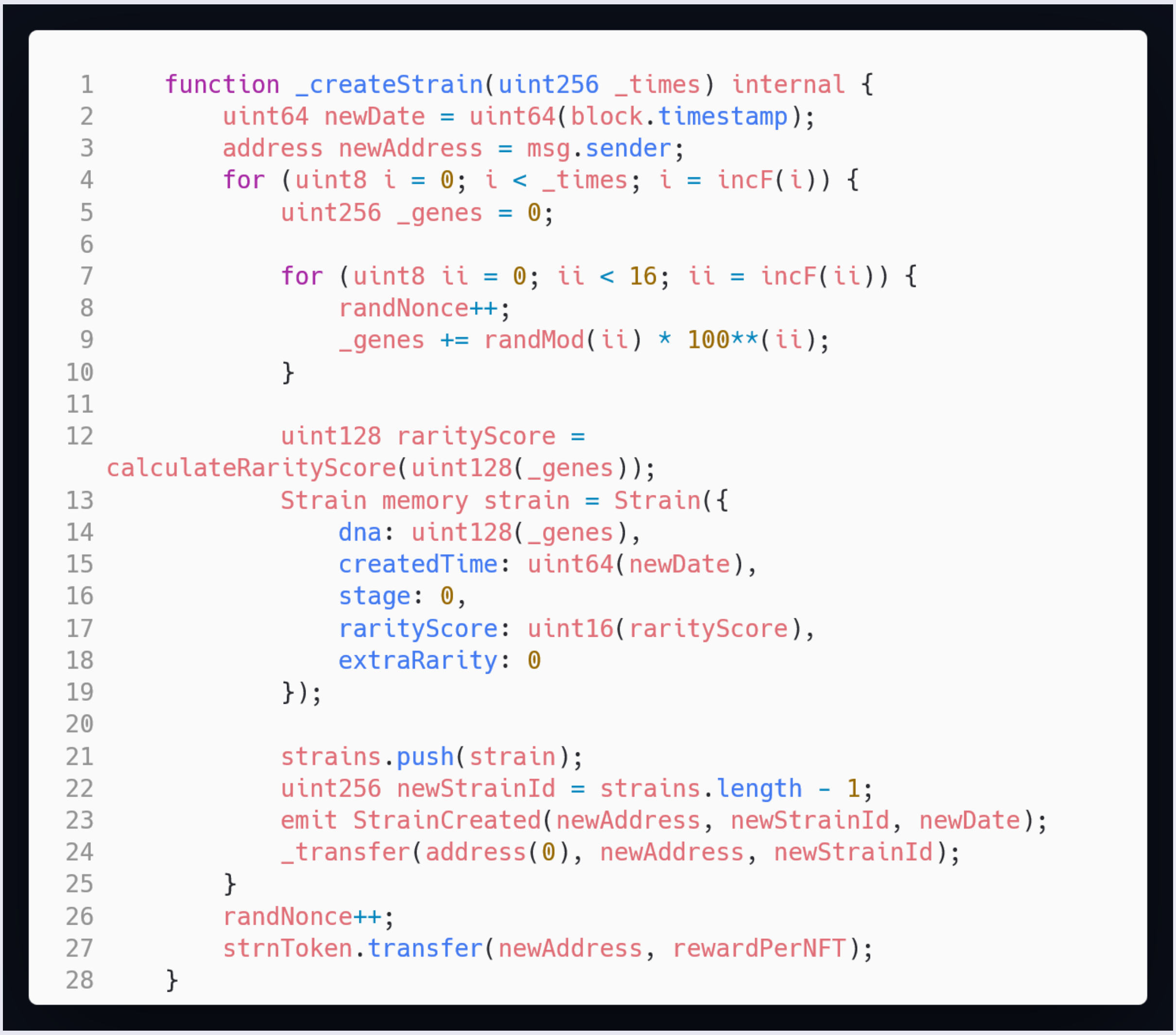
NFT Genes Can Be Manipulated

FINDING ID	#0003
SEVERITY	High Risk
STATUS	Closed
LOCATION	StrainFactory.sol -> 98-116



```
function randMod(uint8 _time) internal view returns (uint8) {
    uint256 rand = uint256(
        keccak256(
            abi.encodePacked(
                block.difficulty,
                block.number,
                block.timestamp,
                block.difficulty,
                msg.sender,
                randNonce,
                _time,
                strains.length
            )
        )
    ) % 100;
    uint256 random = rand / 10;
    uint256 converted = (random == 0) ? 10 : random;
    return uint8(converted);
}
```

LOCATION	StrainFactory.sol -> 130-157



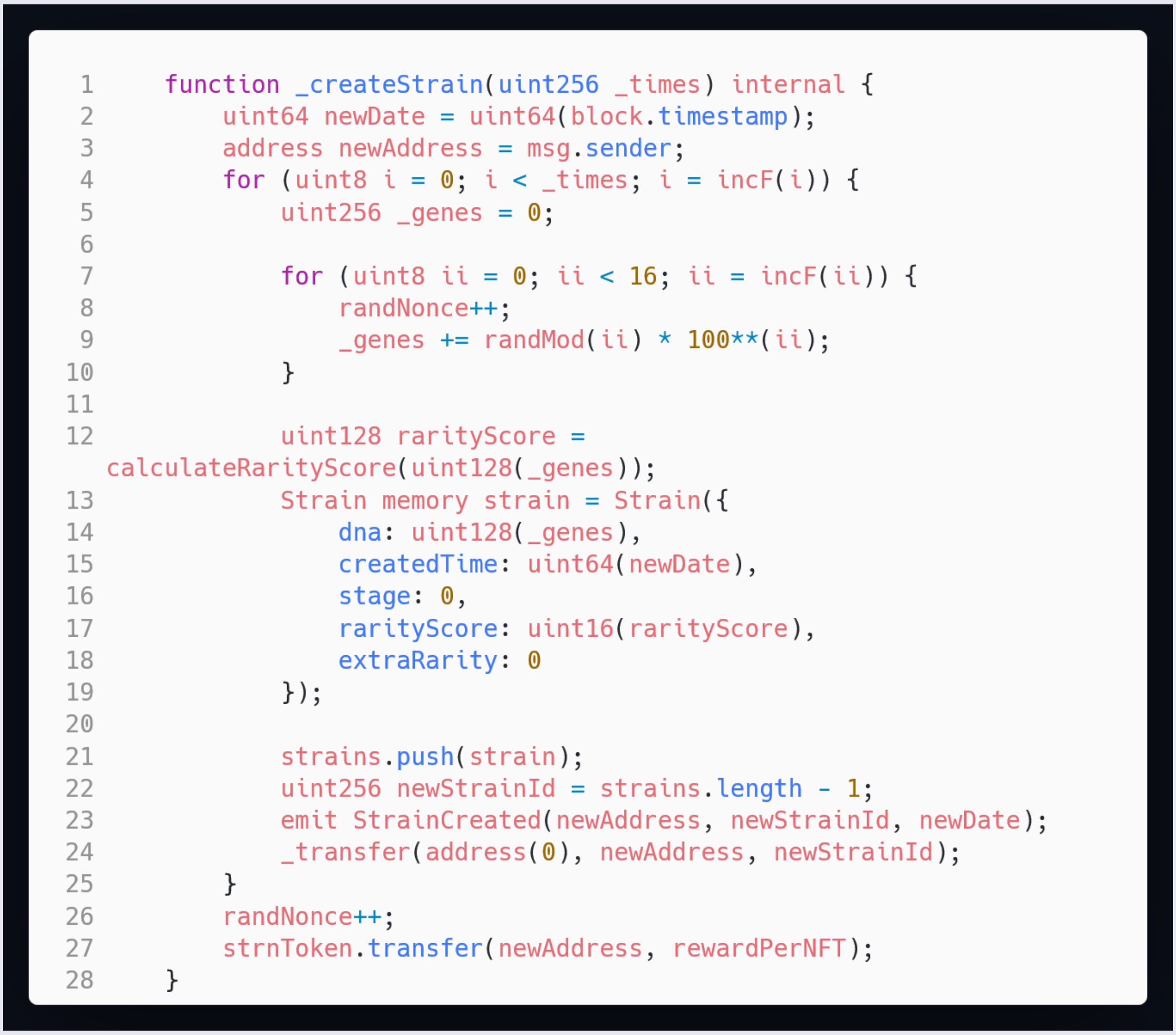
```
    function _createStrain(uint256 _times) internal {
        uint64 newDate = uint64(block.timestamp);
        address newAddress = msg.sender;
        for (uint8 i = 0; i < _times; i = incF(i)) {
            uint256 _genes = 0;

            for (uint8 ii = 0; ii < 16; ii = incF(ii)) {
                randNonce++;
                _genes += randMod(ii) * 100**(ii);
            }

            uint128 rarityScore =
calculateRarityScore(uint128(_genes));
            Strain memory strain = Strain({
                dna: uint128(_genes),
                createdTime: uint64(newDate),
                stage: 0,
                rarityScore: uint16(rarityScore),
                extraRarity: 0
            });

            strains.push(strain);
            uint256 newStrainId = strains.length - 1;
            emit StrainCreated(newAddress, newStrainId, newDate);
            _transfer(address(0), newAddress, newStrainId);
        }
        randNonce++;
        strnToken.transfer(newAddress, rewardPerNFT);
    }
```

DESCRIPTION	Since the pseudorandom function is executed at the time of transaction, a malicious actor can manipulate NFT creation to guarantee desirable qualities such as rarity. This can be accomplished by creating a contract to call *createStrain* and rolling back the transaction if the resulting NFT does not meet specified requirements. This issue becomes even more problematic if the malicious actor is collaborating with the miner, as they can potentially influence some variables in the pseudorandom function. On Fantom: All variables used to calculate randomly the *_genes* of the NFT, are deterministic except for *block.timestamp*. Although, because the block times of Fantom are relatively

	fast and short, only a few values for *block.timestamp* are realistically possible. The malicious actor can narrow down the possible *_genes* that a new strain can get, down to 2-3 values before any other technique.
RECOMMENDATION	We recommend using Chainlink VRF and having the NFT revealed and minted some blocks later to make manipulation more difficult.
RESOLUTION	The project team now makes use of Chainlink, and the reveal of DNA occurs several blocks after the NFT is minted.

Creating Multiple Strains Only Gives Out Reward For A Single One

FINDING ID	#0004
SEVERITY	Medium Risk
STATUS	Closed
LOCATION	StrainFactory.sol -> 130-157

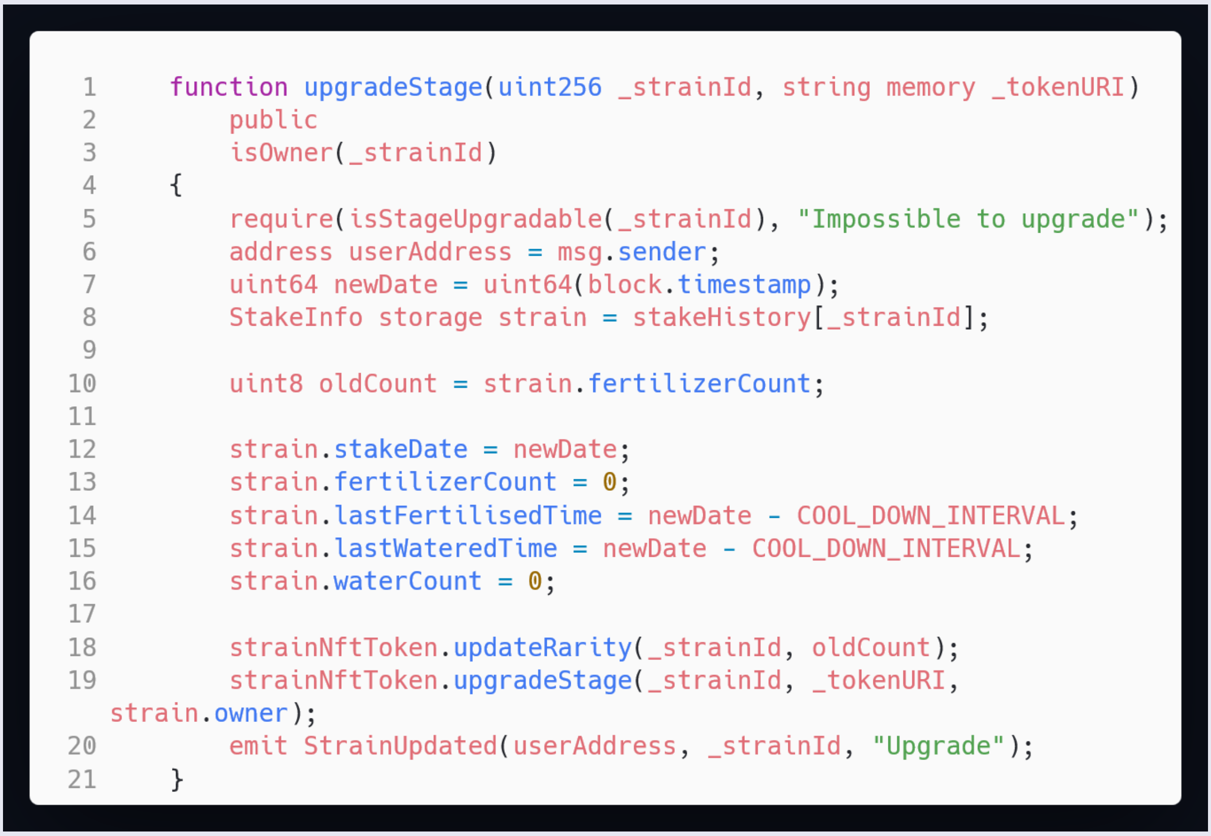


```
    function _createStrain(uint256 _times) internal {
        uint64 newDate = uint64(block.timestamp);
        address newAddress = msg.sender;
        for (uint8 i = 0; i < _times; i = incF(i)) {
            uint256 _genes = 0;

            for (uint8 ii = 0; ii < 16; ii = incF(ii)) {
                randNonce++;
                _genes += randMod(ii) * 100**(ii);
            }

            uint128 rarityScore =
calculateRarityScore(uint128(_genes));
            Strain memory strain = Strain({
                dna: uint128(_genes),
                createdTime: uint64(newDate),
                stage: 0,
                rarityScore: uint16(rarityScore),
                extraRarity: 0
            });

            strains.push(strain);
            uint256 newStrainId = strains.length - 1;
            emit StrainCreated(newAddress, newStrainId, newDate);
            _transfer(address(0), newAddress, newStrainId);
        }
        randNonce++;
        strnToken.transfer(newAddress, rewardPerNFT);
    }
```

DESCRIPTION	When creating multiple strain ERC721 tokens, the user receives only one *rewardPerNFT* quantity of strnTokens, instead of *rewardPerNFT*_times*.
RECOMMENDATION	Calculate the *rewardPerNFT* variable based on the number of tokens minted.
RESOLUTION	The project team has implemented the recommended changes.

Owner Can Choose Arbitrary TokenURI

FINDING ID	#0005
SEVERITY	Medium Risk
STATUS	Closed
LOCATION	StrainNFTStakingContract.sol -> 304-324



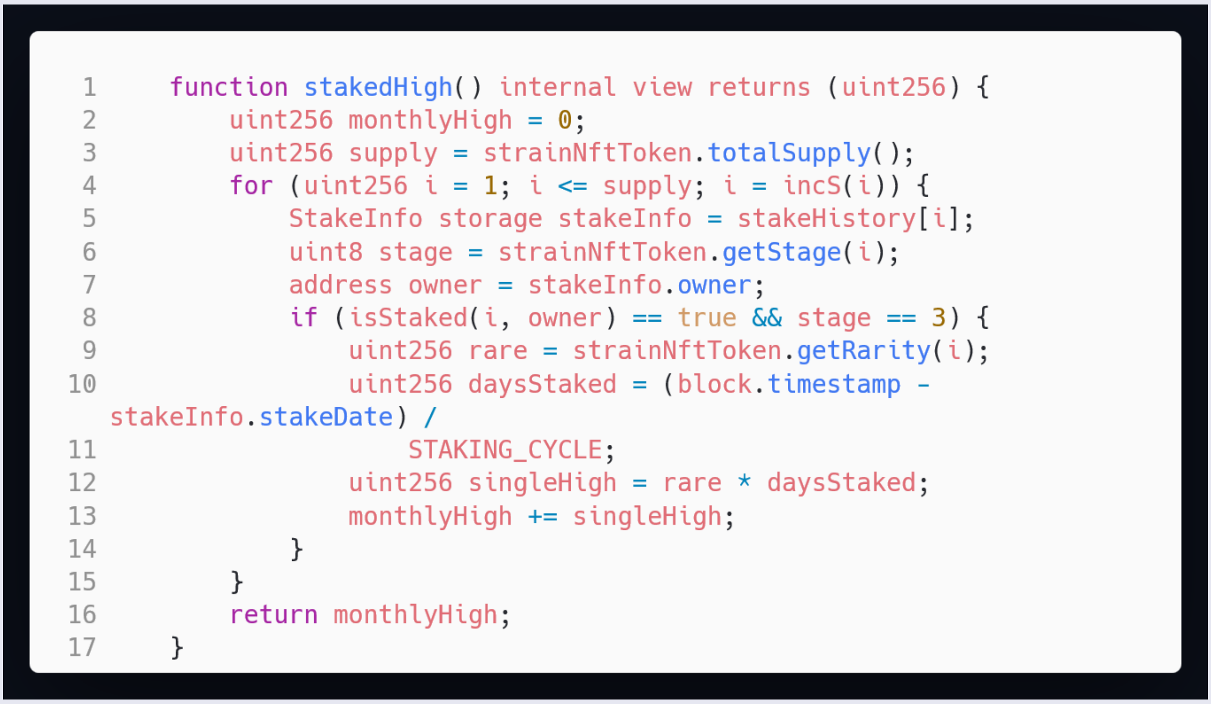
```
    function upgradeStage(uint256 _strainId, string memory _tokenURI)
        public
        isOwner(_strainId)
    {
        require(isStageUpgradable(_strainId), "Impossible to upgrade");
        address userAddress = msg.sender;
        uint64 newDate = uint64(block.timestamp);
        StakeInfo storage strain = stakeHistory[_strainId];

        uint8 oldCount = strain.fertilizerCount;

        strain.stakeDate = newDate;
        strain.fertilizerCount = 0;
        strain.lastFertilisedTime = newDate - COOL_DOWN_INTERVAL;
        strain.lastWateredTime = newDate - COOL_DOWN_INTERVAL;
        strain.waterCount = 0;

        strainNftToken.updateRarity(_strainId, oldCount);
        strainNftToken.upgradeStage(_strainId, _tokenURI, strain.owner);
        emit StrainUpdated(userAddress, _strainId, "Upgrade");
    }
```

DESCRIPTION	The address that owns the ERC721 NFT can choose an arbitrary *tokenURI* when upgrading the stage of their staked strain ERC721 token.
RECOMMENDATION	Implement server-side signature signing to sign a specific *tokenURI* for the user to use or upgrade their NFT on the server side. A common way to use signature signing is \`Signature-Based Minting\`. A more gas-efficient way would be to have a preset IPNS base URI and change the values specific or have a backend on which the tokenURI is hosted (less decentralized than IPNS).
RESOLUTION	The project team removed the possibility of ERC721 owner

	choosing an arbitrary *tokenURI* while upgrading the stage of their NFT. The *tokenURI* is calculated using the DNA, and a fixed URI list.

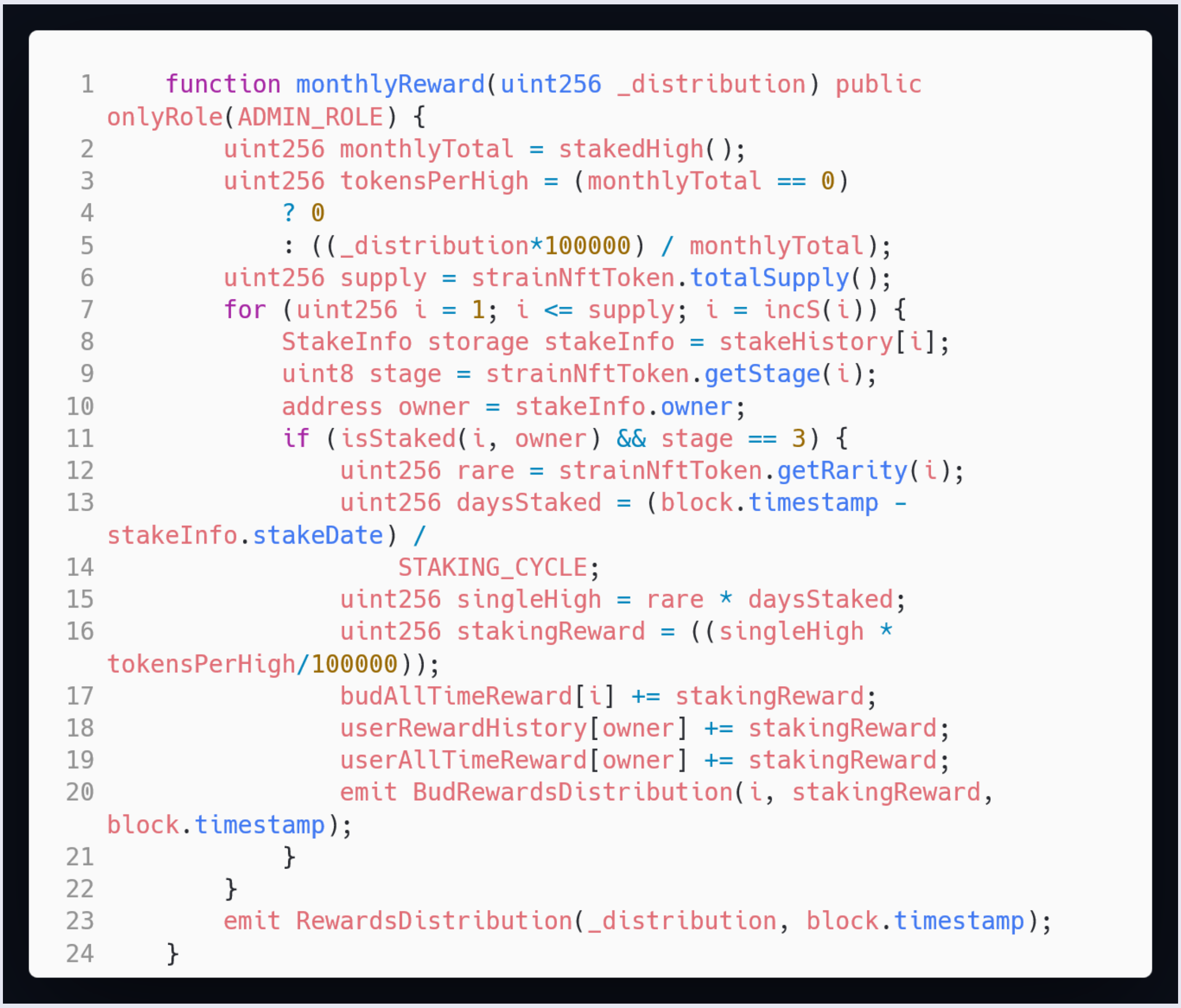
Unbounded Loop

FINDING ID	#0006
SEVERITY	Medium Risk
STATUS	Closed
LOCATION	StrainNFTStakingContract.sol -> 326-342



```
    function stakedHigh() internal view returns (uint256) {
        uint256 monthlyHigh = 0;
        uint256 supply = strainNftToken.totalSupply();
        for (uint256 i = 1; i <= supply; i = incS(i)) {
            StakeInfo storage stakeInfo = stakeHistory[i];
            uint8 stage = strainNftToken.getStage(i);
            address owner = stakeInfo.owner;
            if (isStaked(i, owner) == true && stage == 3) {
                uint256 rare = strainNftToken.getRarity(i);
                uint256 daysStaked = (block.timestamp - stakeInfo.stakeDate) /
                    STAKING_CYCLE;
                uint256 singleHigh = rare * daysStaked;
                monthlyHigh += singleHigh;
            }
        }
        return monthlyHigh;
    }
```

LOCATION	StrainNFTStakingContract.sol -> 344-367



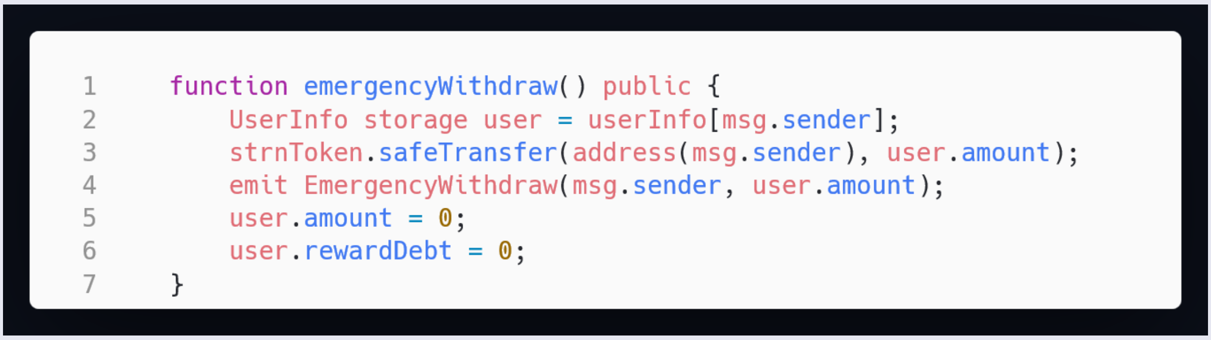
```
    function monthlyReward(uint256 _distribution) public onlyRole(ADMIN_ROLE) {
        uint256 monthlyTotal = stakedHigh();
        uint256 tokensPerHigh = (monthlyTotal == 0)
            ? 0
            : ((_distribution*100000) / monthlyTotal);
        uint256 supply = strainNftToken.totalSupply();
        for (uint256 i = 1; i <= supply; i = incS(i)) {
            StakeInfo storage stakeInfo = stakeHistory[i];
            uint8 stage = strainNftToken.getStage(i);
            address owner = stakeInfo.owner;
            if (isStaked(i, owner) && stage == 3) {
                uint256 rare = strainNftToken.getRarity(i);
                uint256 daysStaked = (block.timestamp - stakeInfo.stakeDate) /
                    STAKING_CYCLE;
                uint256 singleHigh = rare * daysStaked;
                uint256 stakingReward = ((singleHigh * tokensPerHigh/100000));
                budAllTimeReward[i] += stakingReward;
                userRewardHistory[owner] += stakingReward;
                userAllTimeReward[owner] += stakingReward;
                emit BudRewardsDistribution(i, stakingReward, block.timestamp);
            }
        }
        emit RewardsDistribution(_distribution, block.timestamp);
    }
```

DESCRIPTION	Iterating over an unbounded array can cause transactions to revert due to the gas limit. The function *monthlyReward()* runs two O(N) loops where N is the tokenSupply. Depending on the network's max gas limit per transaction, the function could exceed this value due to too many iterations. This function is very expensive to run and there are more gas efficient methods. One such example is to implement a per share-based reward such as in the case of SushiSwap's MasterChef. If the reward rate is time-based, a rebasing mechanism on the reward rate could be used. If the rarity of staked tokens changes, the contract can adjust the reward rates.
RECOMMENDATION	Provide a limit to the size of the array. Alternatively, pass a

	lower and upper index as parameters and iterate over a range.
RESOLUTION	The project team has implemented the recommended changes.

Emergency Withdraw Checks Effects Interactions

FINDING ID	#0007
SEVERITY	Low Risk
STATUS	Closed
LOCATION	• BondingContract.sol -> 117: *function emergencyWithdraw() public {* • FertilizerContract.sol -> 131: *function emergencyWithdraw() public {* • WaterContract.sol -> 131: *function emergencyWithdraw() public {*

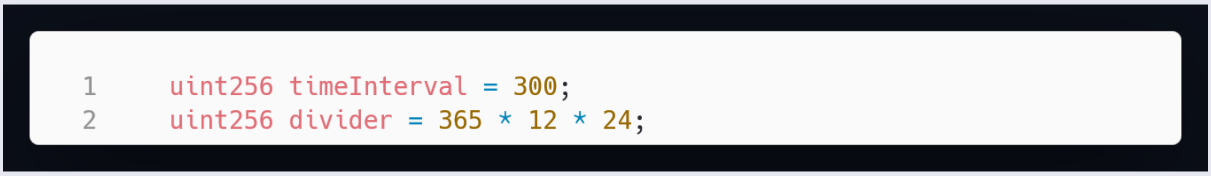


```
function emergencyWithdraw() public {
    UserInfo storage user = userInfo[msg.sender];
    strnToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
}
```

DESCRIPTION	The *emergencyWithdraw()* function violates the Checks-Effects-Interactions pattern. The *lpToken.safeTransfer()* function is executed before setting *user.amount* and *user.rewardDebt* to 0. This could potentially allow a user to drain all the lptokens by re-entering *emergencyWithdraw()* from *lpToken.safeTransfer()*. The likelihood of this happening is low with normal tokens, though if it does occur, it could have severe consequences.
RECOMMENDATION	Modify the function by changing the order of operations such that *user.amount* and *user.rewardDebt* is set to 0 before calling *lpToken.safeTransfer()*.
RESOLUTION	The project team has implemented the recommended changes.

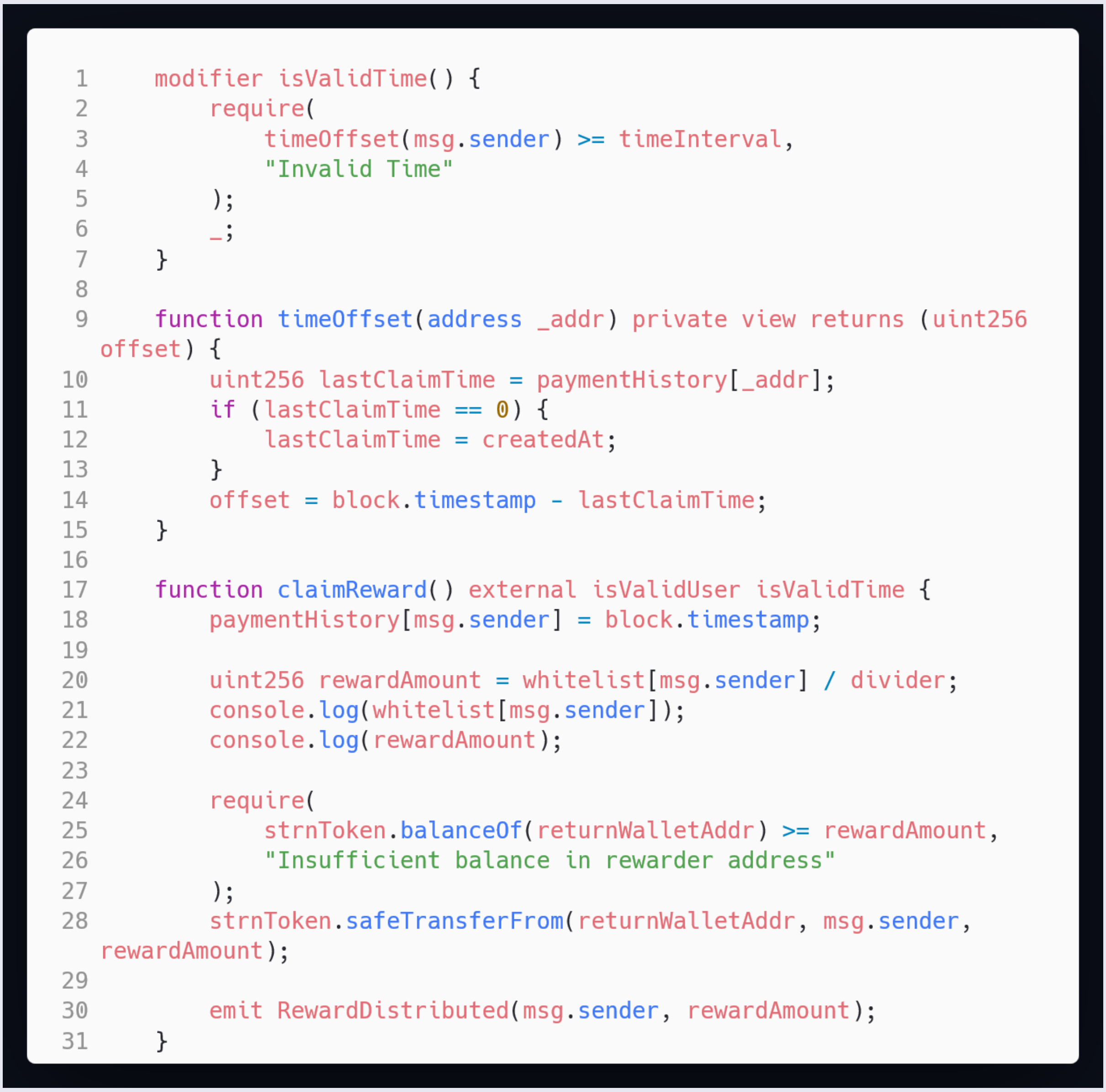
Claiming Rewards Is Time-Based

FINDING ID	#0008
SEVERITY	Low Risk
STATUS	Closed
LOCATION	CommunityReward.sol -> 23-24



```
uint256 timeInterval = 300;
uint256 divider = 365 * 12 * 24;
```

LOCATION	CommunityReward.sol -> 54-84



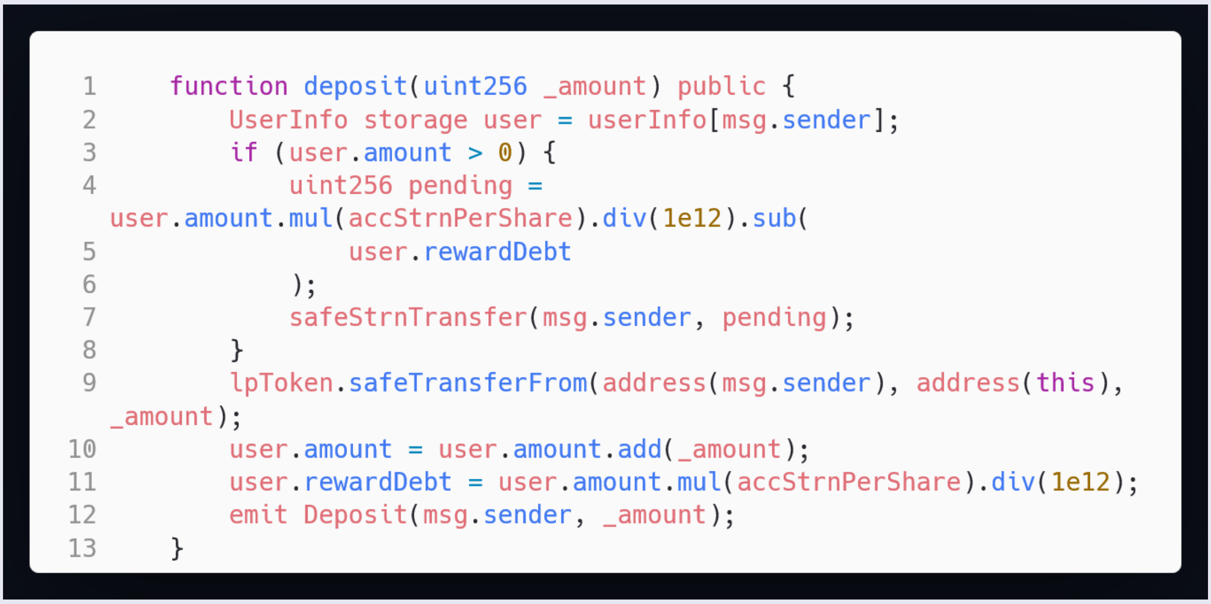
```
modifier isValidTime() {
    require(
        timeOffset(msg.sender) >= timeInterval,
        "Invalid Time"
    );
    _;
}

function timeOffset(address _addr) private view returns (uint256 offset) {
    uint256 lastClaimTime = paymentHistory[_addr];
    if (lastClaimTime == 0) {
        lastClaimTime = createdAt;
    }
    offset = block.timestamp - lastClaimTime;
}

function claimReward() external isValidUser isValidTime {
    paymentHistory[msg.sender] = block.timestamp;

    uint256 rewardAmount = whitelist[msg.sender] / divider;
    console.log(whitelist[msg.sender]);
    console.log(rewardAmount);

    require(
        strnToken.balanceOf(returnWalletAddr) >= rewardAmount,
        "Insufficient balance in rewarder address"
    );
    strnToken.safeTransferFrom(returnWalletAddr, msg.sender, rewardAmount);

    emit RewardDistributed(msg.sender, rewardAmount);
}
```

DESCRIPTION	The user can claim rewards about every 300 seconds. The

	reward the user can claim is constant and equal to his allocation divided by *divider*. As a result, users will have to constantly call the *claimReward()* function to collect their reward.
RECOMMENDATION	Ensure this is the expected behavior, otherwise change the *rewardAmount* to be relative to the last time the user collected his rewards.
RESOLUTION	The *claimReward()* function is now callable only if at least one month has passed since the last call. The *divider* variable now allows claiming only one-twelfth of the total allocation every time *claimReward()* is called.

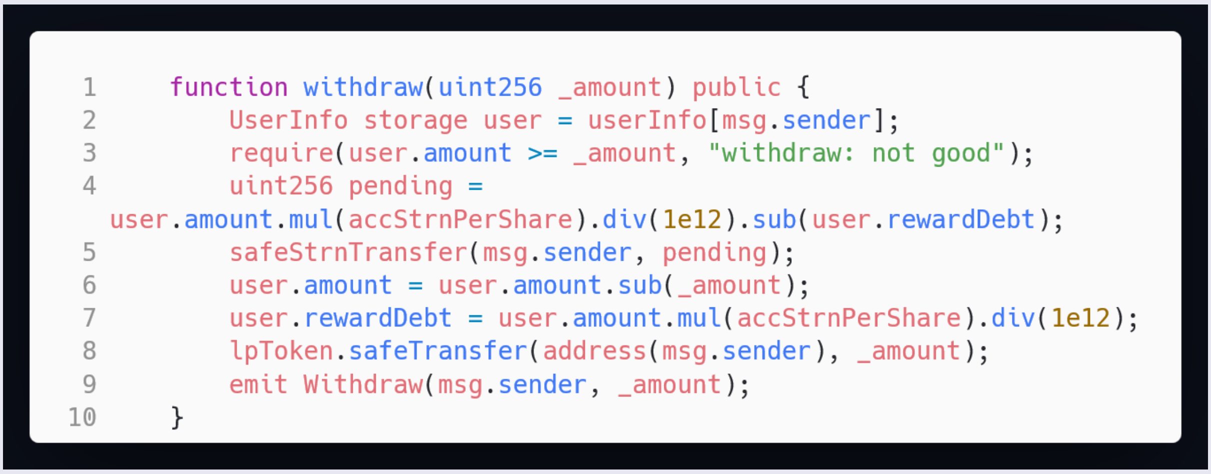
Transferring Zero Rewards

FINDING ID	#0009
SEVERITY	Informational
STATUS	Closed
LOCATION	BondingContract.sol -> 90-102



```
    function deposit(uint256 _amount) public {
        UserInfo storage user = userInfo[msg.sender];
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(accStrnPerShare).div(1e12).sub(
                user.rewardDebt
            );
            safeStrnTransfer(msg.sender, pending);
        }
        lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(accStrnPerShare).div(1e12);
        emit Deposit(msg.sender, _amount);
    }
```

LOCATION	BondingContract.sol -> 105-114



```
    function withdraw(uint256 _amount) public {
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        uint256 pending = user.amount.mul(accStrnPerShare).div(1e12).sub(user.rewardDebt);
        safeStrnTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(accStrnPerShare).div(1e12);
        lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _amount);
    }
```

DESCRIPTION	The call *safeStrnTransfer(msg.sender, pending)* in *deposit()* and *withdraw()* might transfer 0 tokens.
RECOMMENDATION	Avoid this by checking if the pending is greater than zero.
RESOLUTION	The project team has implemented the recommended changes.

No Events Emitted For Changes To Protocol Values

FINDING ID	#0010
SEVERITY	Informational
STATUS	Closed
LOCATION	● BondingContract.sol -> 136: *function setStrnPerBlock(uint256 _strnPerBlock) public onlyOwner {* ● StrainContract.sol -> 473: *function setBaseURI(string memory _uri) internal {* ● StrainContract.sol -> 478: *function setTokenURI(uint256 tokenId, string memory _tokenURI) internal {* ● StrainFactory.sol -> 186: *function updateStrainLimit(uint256 _newLimit) public onlyOwner {* ● StrainFactory.sol -> 200: *function setStakingContract(address _addr) public onlyOwner {*

DESCRIPTION	Functions that change important variables should emit events such that users can more easily monitor the change.
RECOMMENDATION	Emit events from these functions.
RESOLUTION	The project team has implemented the recommended changes.

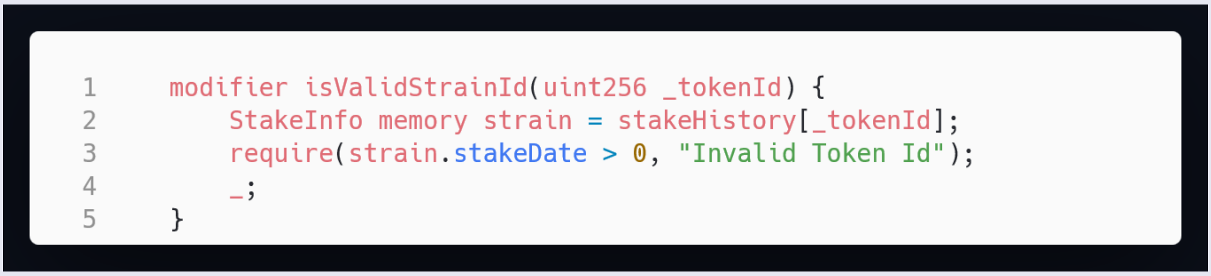
Remove Unnecessary Code

FINDING ID	#0011
SEVERITY	Informational
STATUS	Closed
LOCATION	● CommunityReward.sol -> 8: *import "hardhat/console.sol";* ● CommunityReward.sol -> 74: *console.log(whitelist[msg.sender]);* ● CommunityReward.sol -> 75: *console.log(rewardAmount);* ● StrainNFTStakingContract.sol -> 132: *// strainNftToken.approve(address(this),_tokenId);* ● StrainContract.sol -> 5: *import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";*

DESCRIPTION	These lines are for debugging purposes or are redundant comments.
RECOMMENDATION	Remove these lines.
RESOLUTION	The project team has implemented the recommended changes.

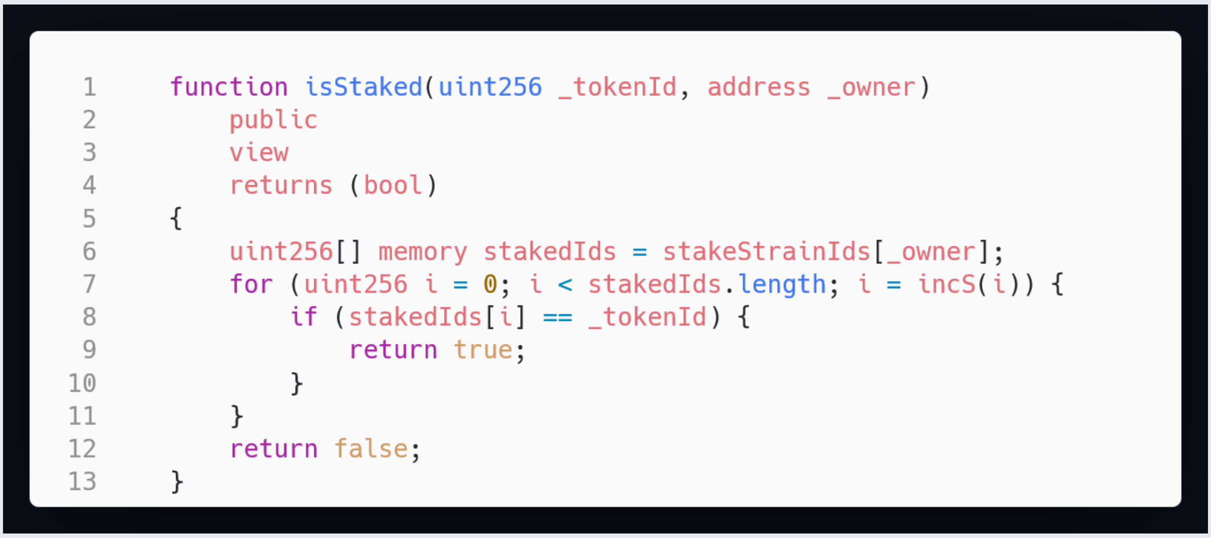
Gas Savings

FINDING ID	#0012
SEVERITY	Informational
STATUS	Open
LOCATION	StrainNFTStakingContract.sol -> 90-94



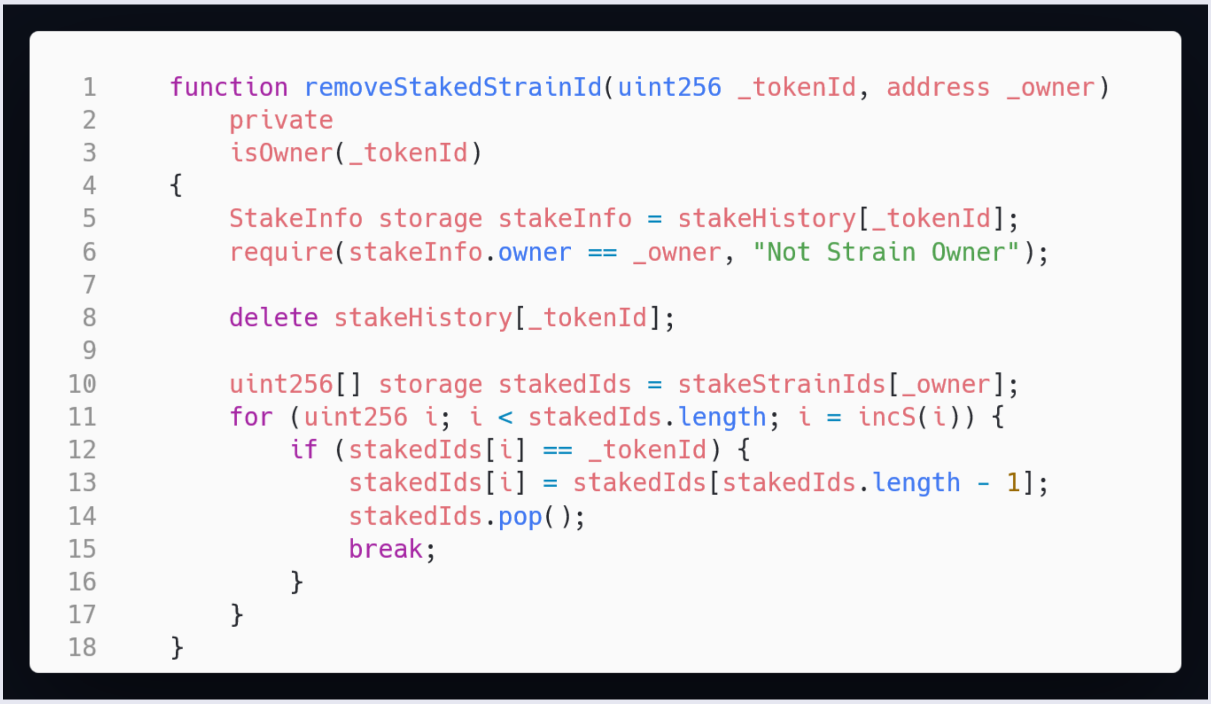
```
modifier isValidStrainId(uint256 _tokenId) {
    StakeInfo memory strain = stakeHistory[_tokenId];
    require(strain.stakeDate > 0, "Invalid Token Id");
    _;
}
```

LOCATION	StrainNFTStakingContract.sol -> 137-149



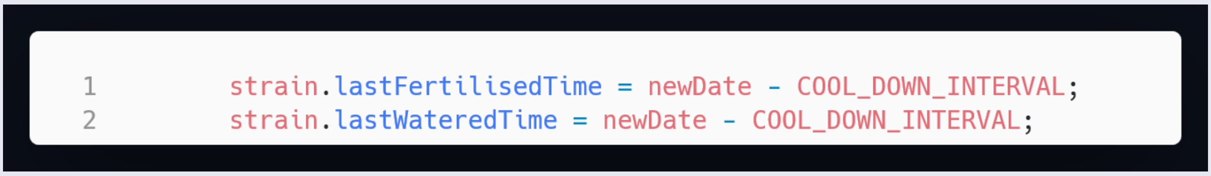
```
function isStaked(uint256 _tokenId, address _owner)
    public
    view
    returns (bool)
{
    uint256[] memory stakedIds = stakeStrainIds[_owner];
    for (uint256 i = 0; i < stakedIds.length; i = incS(i)) {
        if (stakedIds[i] == _tokenId) {
            return true;
        }
    }
    return false;
}
```

LOCATION	StrainNFTStakingContract.sol -> 173-190



```
function removeStakedStrainId(uint256 _tokenId, address _owner)
    private
    isOwner(_tokenId)
{
    StakeInfo storage stakeInfo = stakeHistory[_tokenId];
    require(stakeInfo.owner == _owner, "Not Strain Owner");

    delete stakeHistory[_tokenId];

    uint256[] storage stakedIds = stakeStrainIds[_owner];
    for (uint256 i; i < stakedIds.length; i = incS(i)) {
        if (stakedIds[i] == _tokenId) {
            stakedIds[i] = stakedIds[stakedIds.length - 1];
            stakedIds.pop();
            break;
        }
    }
}
```

LOCATION	StrainNFTStakingContract.sol -> 317-318



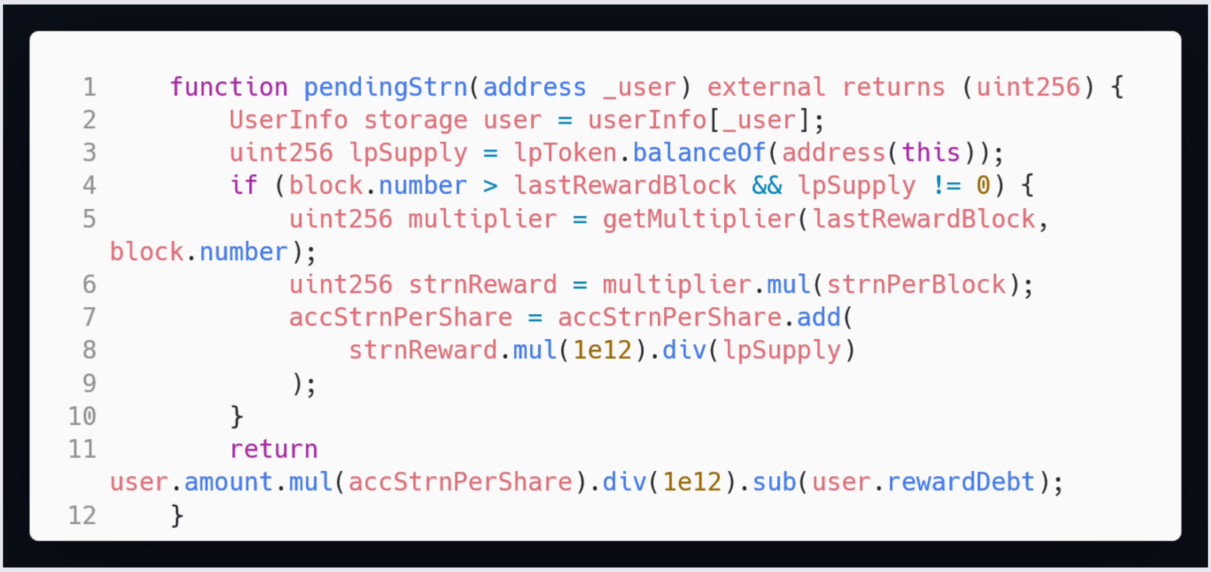
```
    strain.lastFertilisedTime = newDate - COOL_DOWN_INTERVAL;
    strain.lastWateredTime = newDate - COOL_DOWN_INTERVAL;
```

DESCRIPTION	1. Using the *memory* keyword for the *strain* variable consumes a bit more gas, due to unnecessary usage of memory. 2. The functions *stake()*, *stakedHigh()* and *monthlyReward()* call the *isStaked()* function, which costs more the more NFTs the user has staked. 3. The modifier *isOwner()* and the require statement *stakeInfo.owner == _owner*, both check the same condition. 4. In the *upgradeStage()* function, the values of *strain.lastFertilisedTime* and *strain.lastWateredTime* is set so that it can be immediately fertilized/watered. This is unnecessary and has a small gas cost.
RECOMMENDATION	1. Change *memory* to *storage*. 2. That can be prevented by introducing a map of IDs to

	booleans, to check if it is staked or not instantly. This is preferable as looping is very expensive on-chain. 3. One of them can be removed to save on gas. 4. Assign *strain.lastFertilisedTime* and *strain.lastWateredTime* to zero.
RESOLUTION	No changes were made.

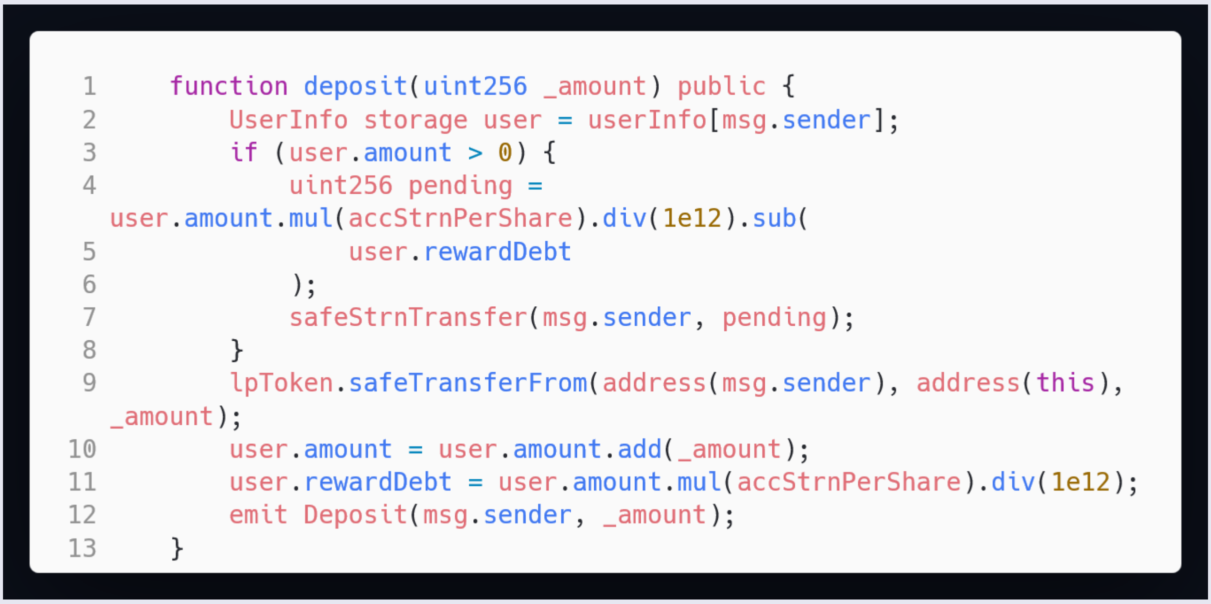
Accumulated Shares Not Updated Correctly

FINDING ID	#0013
SEVERITY	Low Risk
STATUS	Closed
LOCATION	BondingContract.sol -> 76-87



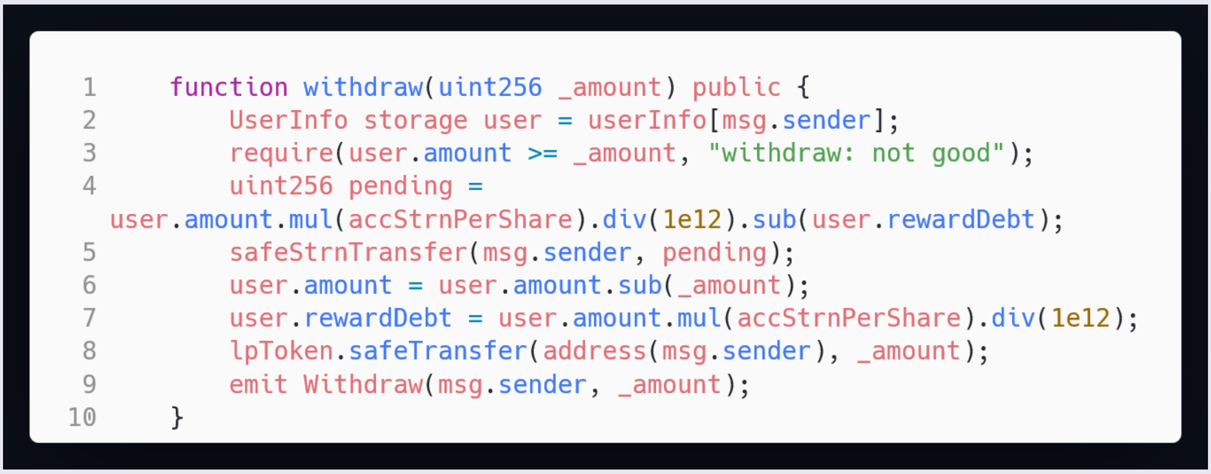
```
    function pendingStrn(address _user) external returns (uint256) {
        UserInfo storage user = userInfo[_user];
        uint256 lpSupply = lpToken.balanceOf(address(this));
        if (block.number > lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(lastRewardBlock, block.number);
            uint256 strnReward = multiplier.mul(strnPerBlock);
            accStrnPerShare = accStrnPerShare.add(
                strnReward.mul(1e12).div(lpSupply)
            );
        }
        return user.amount.mul(accStrnPerShare).div(1e12).sub(user.rewardDebt);
    }
```

LOCATION	BondingContract.sol -> 90-102



```
    function deposit(uint256 _amount) public {
        UserInfo storage user = userInfo[msg.sender];
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(accStrnPerShare).div(1e12).sub(
                user.rewardDebt
            );
            safeStrnTransfer(msg.sender, pending);
        }
        lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(accStrnPerShare).div(1e12);
        emit Deposit(msg.sender, _amount);
    }
```

LOCATION	BondingContract.sol -> 105-114

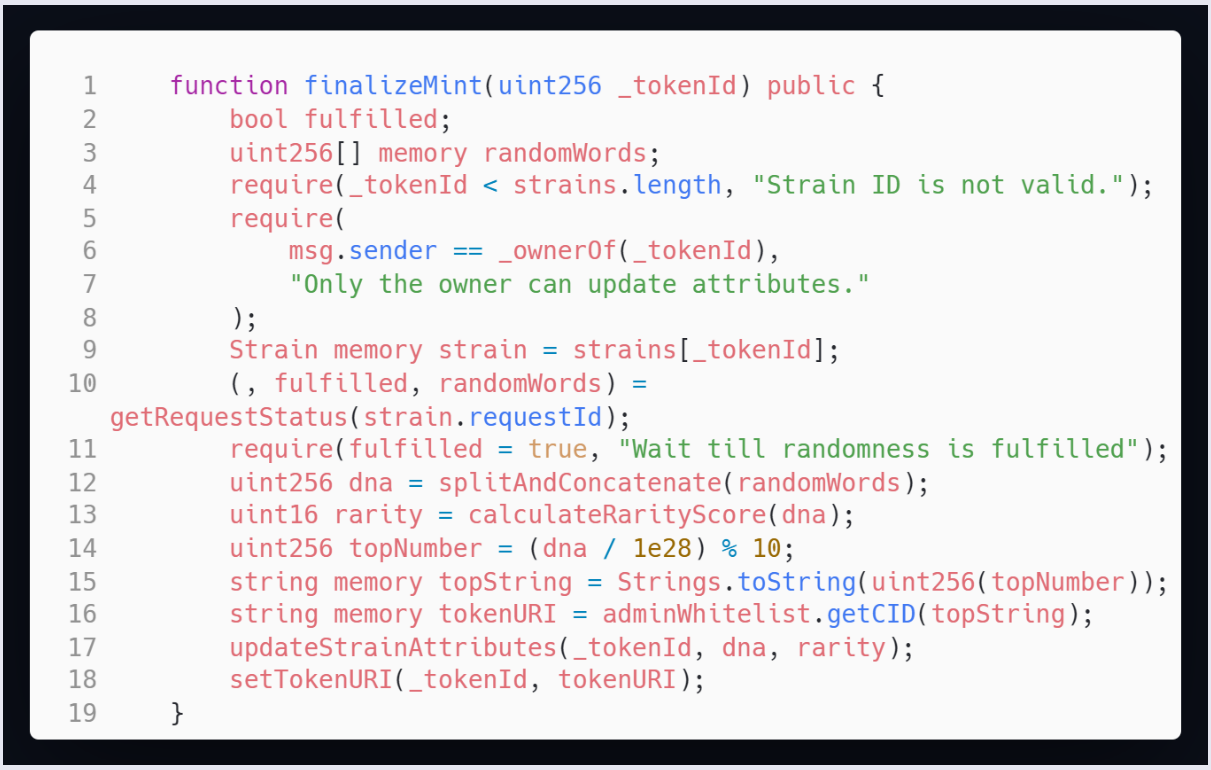


```
    function withdraw(uint256 _amount) public {
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        uint256 pending =
user.amount.mul(accStrnPerShare).div(1e12).sub(user.rewardDebt);
        safeStrnTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(accStrnPerShare).div(1e12);
        lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _amount);
    }
```

DESCRIPTION	The function *pendingStrn()* should be called when a *deposit()* or *withdraw()* happens, in order to correctly reward the users based on their share of the reward. Currently, because *lastRewardBlock* is not updated, the user can drain all the rewards by calling *pendingStrn()* repeatedly. Additionally, the variables *lpToken* and *lastRewardBlock* are not initialized. As a result, the contract functionality is broken.
RECOMMENDATION	Initialize the above variables. Change *pendingStrn()* to a view function, and introduce an *update()* function that is called at *deposit()* and *withdraw()*.
RESOLUTION	The project team has implemented the recommended changes.

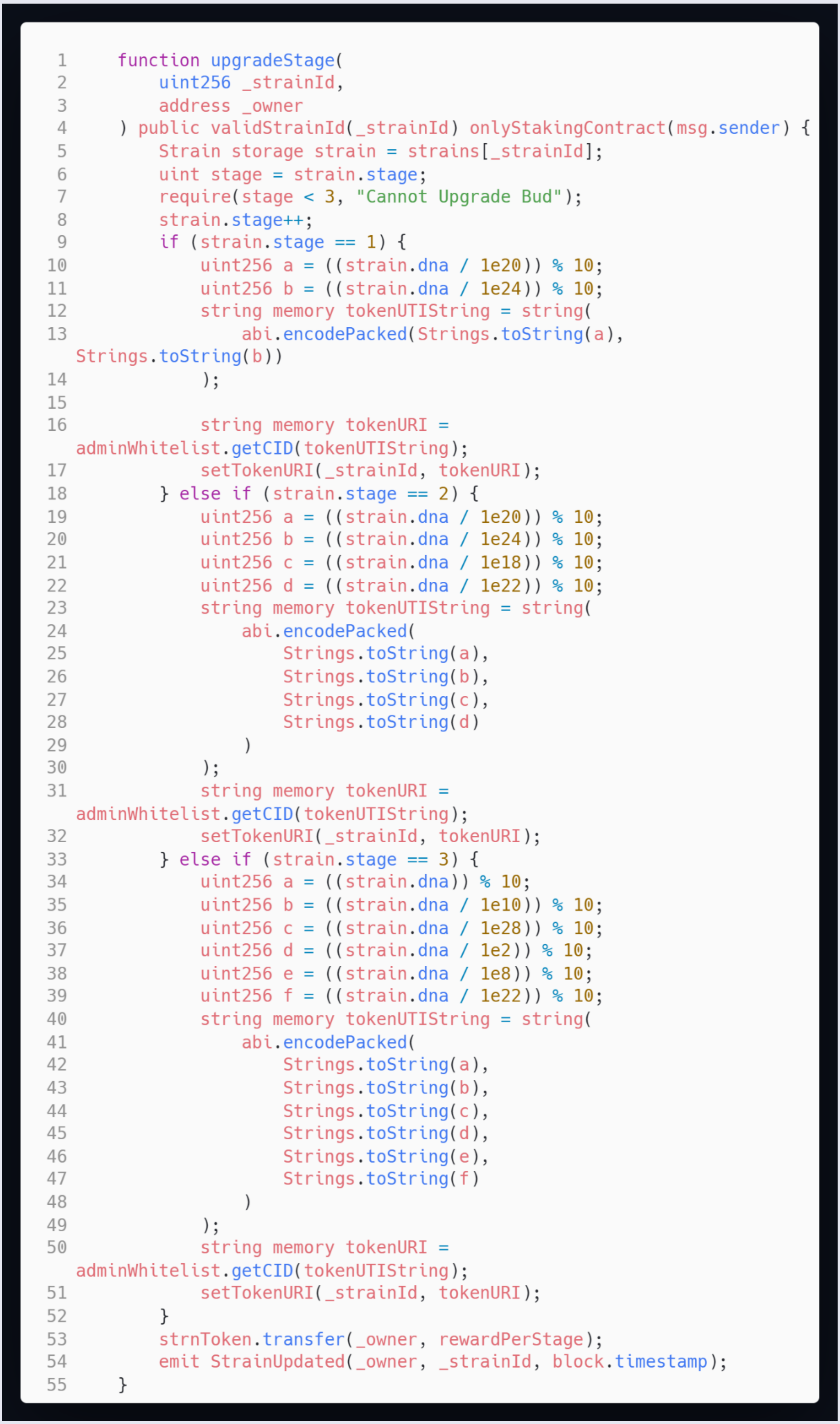
TokenURI Can Be Modified By Whitelisted Admins

FINDING ID	#0014
SEVERITY	Low Risk
STATUS	Open
LOCATION	StrainFactory.sol -> 148-166



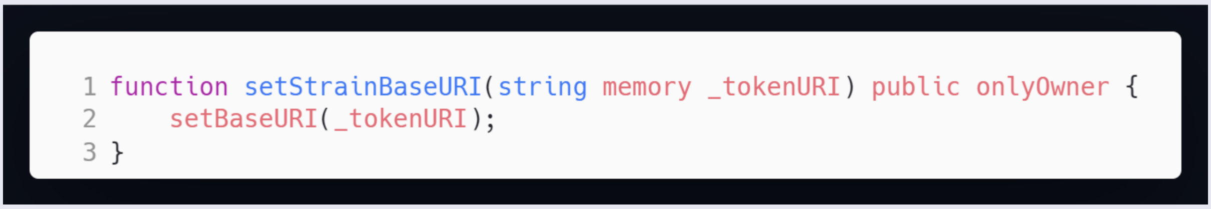
```
    function finalizeMint(uint256 _tokenId) public {
        bool fulfilled;
        uint256[] memory randomWords;
        require(_tokenId < strains.length, "Strain ID is not valid.");
        require(
            msg.sender == _ownerOf(_tokenId),
            "Only the owner can update attributes."
        );
        Strain memory strain = strains[_tokenId];
        (, fulfilled, randomWords) =
getRequestStatus(strain.requestId);
        require(fulfilled = true, "Wait till randomness is fulfilled");
        uint256 dna = splitAndConcatenate(randomWords);
        uint16 rarity = calculateRarityScore(dna);
        uint256 topNumber = (dna / 1e28) % 10;
        string memory topString = Strings.toString(uint256(topNumber));
        string memory tokenURI = adminWhitelist.getCID(topString);
        updateStrainAttributes(_tokenId, dna, rarity);
        setTokenURI(_tokenId, tokenURI);
    }
```

LOCATION	StrainFactory.sol -> 243-297



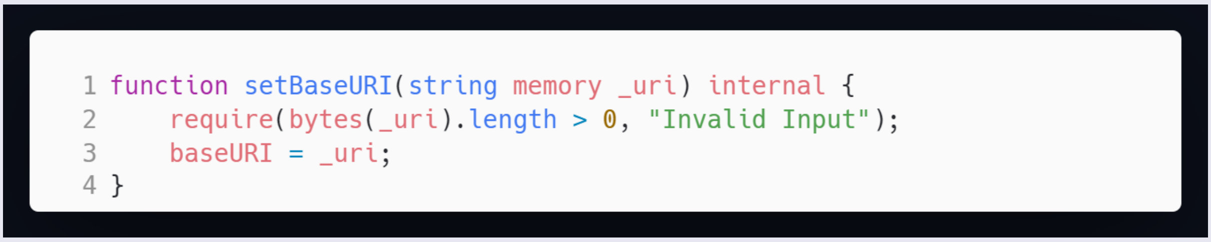
```solidity
    function upgradeStage(
        uint256 _strainId,
        address _owner
    ) public validStrainId(_strainId) onlyStakingContract(msg.sender) {
        Strain storage strain = strains[_strainId];
        uint stage = strain.stage;
        require(stage < 3, "Cannot Upgrade Bud");
        strain.stage++;
        if (strain.stage == 1) {
            uint256 a = ((strain.dna / 1e20)) % 10;
            uint256 b = ((strain.dna / 1e24)) % 10;
            string memory tokenUTIString = string(
                abi.encodePacked(Strings.toString(a), Strings.toString(b))
            );

            string memory tokenURI = adminWhitelist.getCID(tokenUTIString);
            setTokenURI(_strainId, tokenURI);
        } else if (strain.stage == 2) {
            uint256 a = ((strain.dna / 1e20)) % 10;
            uint256 b = ((strain.dna / 1e24)) % 10;
            uint256 c = ((strain.dna / 1e18)) % 10;
            uint256 d = ((strain.dna / 1e22)) % 10;
            string memory tokenUTIString = string(
                abi.encodePacked(
                    Strings.toString(a),
                    Strings.toString(b),
                    Strings.toString(c),
                    Strings.toString(d)
                )
            );
            string memory tokenURI = adminWhitelist.getCID(tokenUTIString);
            setTokenURI(_strainId, tokenURI);
        } else if (strain.stage == 3) {
            uint256 a = ((strain.dna)) % 10;
            uint256 b = ((strain.dna / 1e10)) % 10;
            uint256 c = ((strain.dna / 1e28)) % 10;
            uint256 d = ((strain.dna / 1e2)) % 10;
            uint256 e = ((strain.dna / 1e8)) % 10;
            uint256 f = ((strain.dna / 1e22)) % 10;
            string memory tokenUTIString = string(
                abi.encodePacked(
                    Strings.toString(a),
                    Strings.toString(b),
                    Strings.toString(c),
                    Strings.toString(d),
                    Strings.toString(e),
                    Strings.toString(f)
                )
            );
            string memory tokenURI = adminWhitelist.getCID(tokenUTIString);
            setTokenURI(_strainId, tokenURI);
        }
        strnToken.transfer(_owner, rewardPerStage);
        emit StrainUpdated(_owner, _strainId, block.timestamp);
    }
```

LOCATION	StrainFactory.Sol -> 239-241



```
function setStrainBaseURI(string memory _tokenURI) public onlyOwner {
    setBaseURI(_tokenURI);
}
```

LOCATION	StrainContract.sol -> 352-355

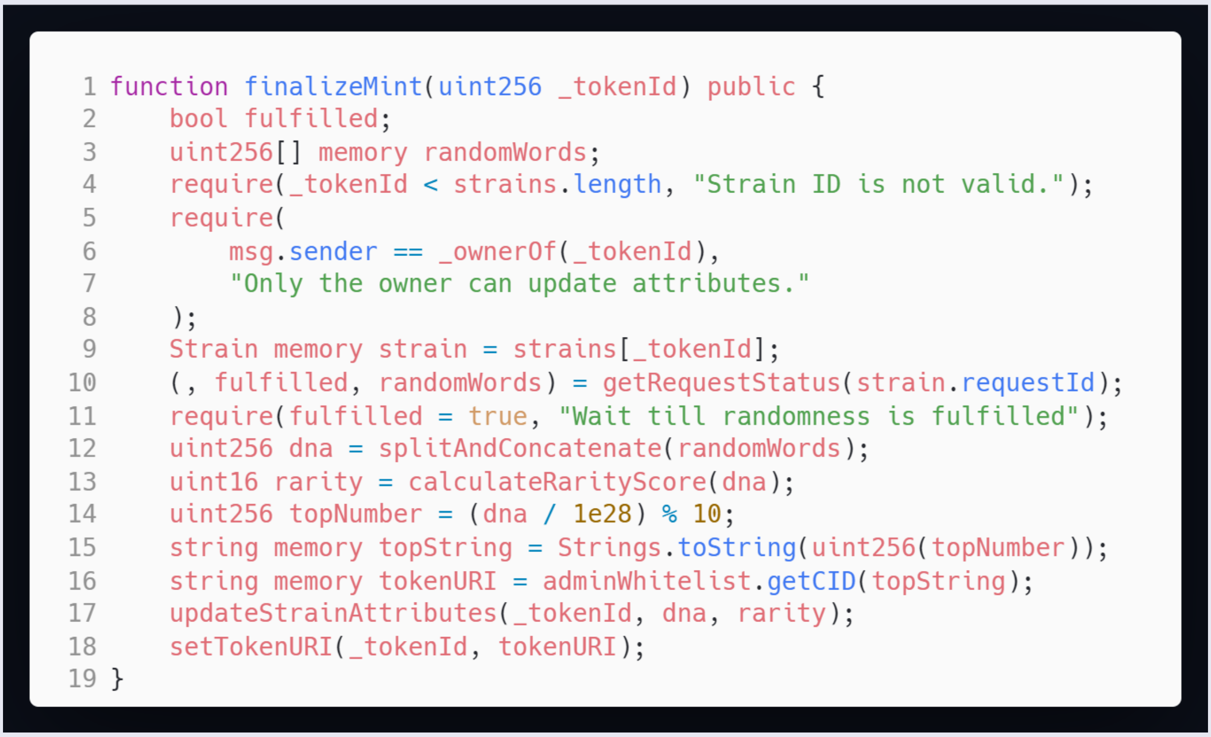


```
function setBaseURI(string memory _uri) internal {
    require(bytes(_uri).length > 0, "Invalid Input");
    baseURI = _uri;
}
```

DESCRIPTION	The *AdminWhitelist* contract sets the tokenURIs based on a string. The contract owner is the first one who can change these tokenURIs. There is also functionality to add more whitelist owners to the contract. Also, the owner of *StrainFactory* can set the *baseURI* of *StrainContract.* If the owner is compromised, then the baseURI or tokenURI could be swapped with malicious intent. This could change token metadata and token images.
RECOMMENDATION	Pre-publish the NFTs and select them via randomness. For example, you could have all tokenURIs in a list, select one using the random number, and delete it from the list. Another example would be to use the stats to fetch a specific token from a list. So even if the DNA is different, the stats could fetch the same image and stats. Later on, if it's upgraded its image could change.
RESOLUTION	No changes were made.

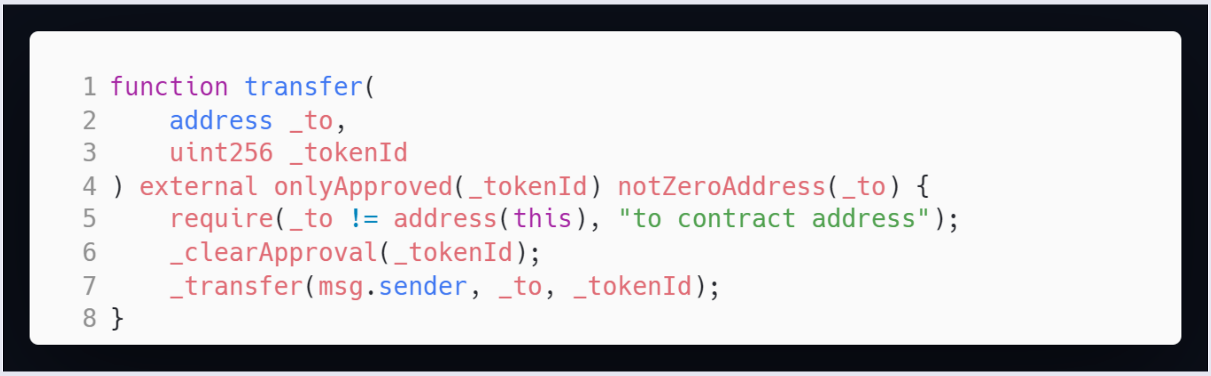
NFTs Can Be Transferred Without Finalizing The Mint

FINDING ID	#0015
SEVERITY	Low Risk
STATUS	Closed
LOCATION	StrainFactory.sol -> 148-166



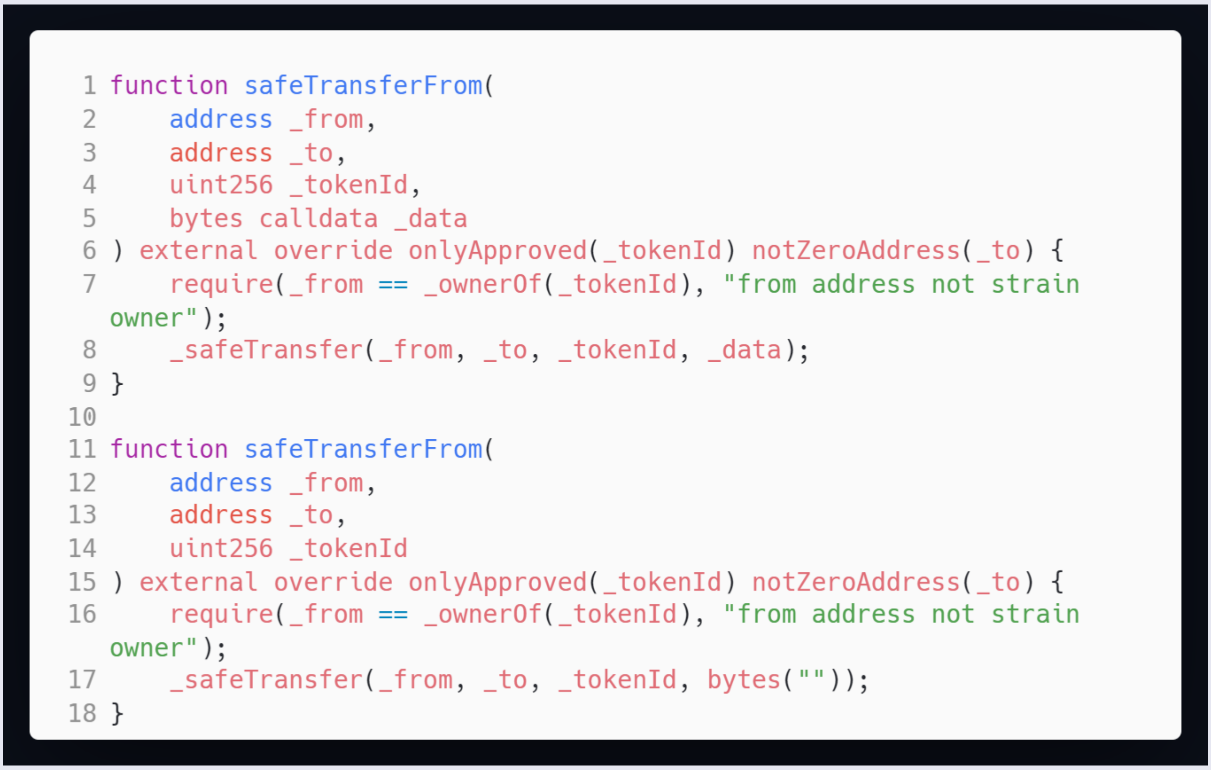
```
function finalizeMint(uint256 _tokenId) public {
    bool fulfilled;
    uint256[] memory randomWords;
    require(_tokenId < strains.length, "Strain ID is not valid.");
    require(
        msg.sender == _ownerOf(_tokenId),
        "Only the owner can update attributes."
    );
    Strain memory strain = strains[_tokenId];
    (, fulfilled, randomWords) = getRequestStatus(strain.requestId);
    require(fulfilled = true, "Wait till randomness is fulfilled");
    uint256 dna = splitAndConcatenate(randomWords);
    uint16 rarity = calculateRarityScore(dna);
    uint256 topNumber = (dna / 1e28) % 10;
    string memory topString = Strings.toString(uint256(topNumber));
    string memory tokenURI = adminWhitelist.getCID(topString);
    updateStrainAttributes(_tokenId, dna, rarity);
    setTokenURI(_tokenId, tokenURI);
}
```

LOCATION	StrainContract.sol -> 179-186



```
function transfer(
    address _to,
    uint256 _tokenId
) external onlyApproved(_tokenId) notZeroAddress(_to) {
    require(_to != address(this), "to contract address");
    _clearApproval(_tokenId);
    _transfer(msg.sender, _to, _tokenId);
}
```

LOCATION	StrainContract.sol -> 292-309



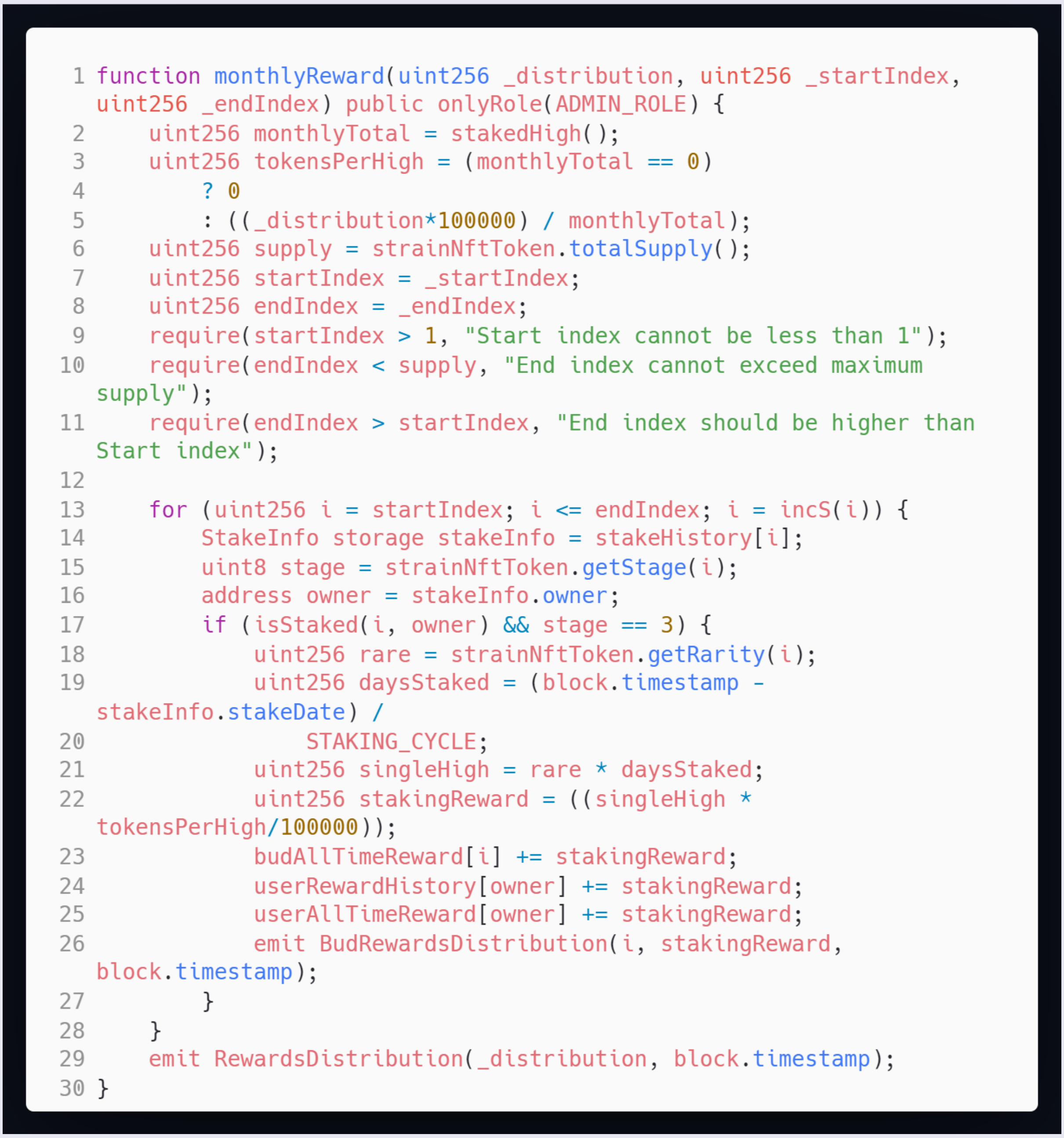
```
function safeTransferFrom(
    address _from,
    address _to,
    uint256 _tokenId,
    bytes calldata _data
) external override onlyApproved(_tokenId) notZeroAddress(_to) {
    require(_from == _ownerOf(_tokenId), "from address not strain owner");
    _safeTransfer(_from, _to, _tokenId, _data);
}

function safeTransferFrom(
    address _from,
    address _to,
    uint256 _tokenId
) external override onlyApproved(_tokenId) notZeroAddress(_to) {
    require(_from == _ownerOf(_tokenId), "from address not strain owner");
    _safeTransfer(_from, _to, _tokenId, bytes(""));
}
```

DESCRIPTION	When a user mints an NFT, a Chainlink request is initiated to generate a random number. To "finalize" the minting process, the user must invoke *finalizeMint*. However, once the Chainlink request is completed, even if *finalizeMint* hasn't been called, it's possible to infer the outcome of the mint. Given that the NFT can be transferred without invoking the *finalizeMint* function, someone might sell an NFT with undesirable attributes as “unminted”.
RECOMMENDATION	We recommend locking down transfers until the minting is finalized or using Chainlink's callback to finalize the mint. Note: If you opt to use the *finalizeMint* function with Chainlink's callback, ensure that the callback cannot fail. Proving that this won't fail can be complex as it's influenced by the gas considerations set in the Chainlink VRF Consumer.
RESOLUTION	While the modifications to the transfer function work, it's not advisable to alter the transfer function in this manner. Doing so will result in additional gas costs for every transfer, rather than just at minting. An alternative solution might have been to withhold the token transfer

	from the factory until it was finalized.

Rewards Distribution Is Callable Only By The Admin Role

FINDING ID	#0016
SEVERITY	Medium Risk
STATUS	Open
LOCATION	StrainNFTStakingContract.sol -> 324-353



```
function monthlyReward(uint256 _distribution, uint256 _startIndex, uint256 _endIndex) public onlyRole(ADMIN_ROLE) {
    uint256 monthlyTotal = stakedHigh();
    uint256 tokensPerHigh = (monthlyTotal == 0)
        ? 0
        : ((_distribution*100000) / monthlyTotal);
    uint256 supply = strainNftToken.totalSupply();
    uint256 startIndex = _startIndex;
    uint256 endIndex = _endIndex;
    require(startIndex > 1, "Start index cannot be less than 1");
    require(endIndex < supply, "End index cannot exceed maximum supply");
    require(endIndex > startIndex, "End index should be higher than Start index");

    for (uint256 i = startIndex; i <= endIndex; i = incS(i)) {
        StakeInfo storage stakeInfo = stakeHistory[i];
        uint8 stage = strainNftToken.getStage(i);
        address owner = stakeInfo.owner;
        if (isStaked(i, owner) && stage == 3) {
            uint256 rare = strainNftToken.getRarity(i);
            uint256 daysStaked = (block.timestamp - stakeInfo.stakeDate) /
                STAKING_CYCLE;
            uint256 singleHigh = rare * daysStaked;
            uint256 stakingReward = ((singleHigh * tokensPerHigh/100000));
            budAllTimeReward[i] += stakingReward;
            userRewardHistory[owner] += stakingReward;
            userAllTimeReward[owner] += stakingReward;
            emit BudRewardsDistribution(i, stakingReward, block.timestamp);
        }
    }
    emit RewardsDistribution(_distribution, block.timestamp);
}
```

DESCRIPTION	The *monthlyReward* function is only callable by the admin, and also, the admin can select the indexes of the token IDs that are going to get their rewards. This means that the admin can call this function and drain

	the rewards by calling the *monthlyReward* for a subset of tokens. The admin could also withhold rewards on a subset of tokens (there is no guarantee that the rewards are going to be distributed).
RECOMMENDATION	The priority is to resolve the drain problem, this can be done by restricting the number of times rewards can be claimed for a token. We recommend that A. Add additional information in StakeInfo, which will indicate the last time this token ID got its reward, and it will be restricted to getting rewards more than once every 30 days. (This will prohibit the admin from draining the rewards) B. Additionally to Point A, make this function public (each token owner could call this function to get their reward if needed).
RESOLUTION	No changes were made.

On-Chain Analysis

Not Analyzed Yet

External Addresses

Externally Owned Accounts

Owner

ACCOUNT	Address
USAGE	0x123456... *Contract.owner* - Variable
IMPACT	● receives elevated permissions as owner, operator, or other

Template

ACCOUNT	Address
USAGE	0x123456... *Contract.owner* - Variable
IMPACT	● receives transfer of tokens deposited by users ● receives allowance of tokens deposited by users ● receives transfer of tokens deposited or minted by project ● receives allowance of tokens deposited or minted by project ● impacts ability to deposit or withdraw tokens ● impacts other user actions ● impacts owner/operator actions ● receives elevated permissions as owner, operator, or other ● provides proxy function implementations

External Contracts

These contracts are not part of the audit scope.

Some Vault

ADDRESS	ETH - 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
USAGE	0x123456... *SomeContract.Vault* - Constant
IMPACT	● ERC20 Token

Template

ADDRESS	ETH - 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
USAGE	0x123456... *SomeContract.WETH* - Constant *SomeContract.want* - Immutable 0xfedcba... *OtherContract.pool[0].lpToken* - Set Once *OtherContract.rewardToken* - Variable
IMPACT	● ERC20 Token ● ERC721 Token (Non-Fungible Token) ● Timelock ● MasterChef ● Uniswap Router ● Uniswap Liquidity Pool ● Uniswap Oracle ● Chainlink Price Feed ● Chainlink VRF Coordinator ● receives transfer of tokens deposited by users ● receives allowance of tokens deposited by users ● receives transfer of tokens deposited or minted by project ● receives allowance of tokens deposited or minted by project ● impacts ability to deposit or withdraw tokens ● impacts other user actions ● impacts owner/operator actions ● has elevated permissions as owner, operator, or other

	● provides proxy function implementations

External Tokens

These contracts are not part of the audit scope.

Wrapped Ether

ADDRESS	ETH - 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
USAGE	0x123456... *SomeContract.WETH* - Constant
IMPACT	● ERC20 Token

Appendix A - Reviewed Documents

Deployed Contracts

Document	Address
AdminWhitelist.sol	N/A
BondingContract.sol	N/A
FertilizerContract.sol	N/A
WateringContract.sol	N/A
CommunityReward.sol	N/A
FERT.sol	N/A
StrainContract.sol	N/A
StrainFactory.sol	N/A
StrainNFTStakingContract.sol	N/A
STRN.sol	N/A
WTR.sol	N/A

Libraries And Interfaces

Revisions

Revision 1	Solidity 0.6.12 6df22abf61215edebf46abbfd4a9a563fac1ba87 Solidity 0.8.12 c663214e350d38226f501be08f383e75da8925e1

Imported Contracts

OpenZeppelin	Solidity 0.6.12 - OpenZeppelin 3.4.0 Solidity 0.8.12 - OpenZeppelin 4.5.0

Appendix B - Risk Ratings

Risk	Description
High Risk	Security risks that are ***almost certain*** to lead to ***impairment or loss of funds***. Projects are advised to fix as soon as possible.
Medium Risk	Security risks that are ***very likely*** to lead to ***impairment or loss of funds*** with ***limited impact***. Projects are advised to fix as soon as possible.
Low Risk	Security risks that can lead to ***damage to the protocol***. Projects are advised to fix. Issues with this rating might be used in an exploit with other issues to cause significant damage.
Informational	Noteworthy information. Issues may include code conventions, missing or conflicting information, gas optimizations, and other advisories.

Appendix C - Finding Statuses

Closed	Contracts were modified to permanently resolve the finding.
Mitigated	The finding was resolved on-chain. The issue may require monitoring, for example in the case of a time lock.
Partially Closed	Contracts were modified to partially fix the issue
Partially Mitigated	The finding was resolved by project specific methods which cannot be verified on chain. Examples include compounding at a given frequency, or the use of a multisig wallet.
Open	The finding was not addressed.

Appendix D - Glossary

Contract Structure

Contract: An address that provides functionality to users and other contracts. They are implemented in code and deployed to the blockchain.
Protocol: A system of contracts that work together.
Stakeholders: The users, operators, owners, and other participants of a contract.

Security Concepts

***Bug*:** A defect in the contract code.
***Exploit*:** A chain of events involving bugs, vulnerabilities, or other security risks that damages a protocol.
Funds: Tokens deposited by users or other stakeholders into a protocol.
Impairment: The loss of functionality in a contract or protocol.
Security risk: A circumstance that may result in harm to the stakeholders of a protocol. Examples include vulnerabilities in the code, bugs, excessive permissions, missing timelock, etc.
Vulnerability: A vulnerability is a flaw that allows an attacker to potentially cause harm to the stakeholders of a contract. They may occur in a contract's code, design, or deployed state on the blockchain.

Appendix E - Audit Procedure

A typical Obelisk audit uses a combination of the three following methods:

Manual analysis consists of a direct inspection of the contracts to identify any security issues. Obelisk auditors use their experience in software development to spot vulnerabilities. Their familiarity with common contracts allows them to identify a wide range of issues in both forked contracts as well as original code.

Static analysis is software analysis of the contracts. Such analysis is called “static” as it examines the code outside of a runtime environment. Static analysis is a powerful tool used by auditors to identify subtle issues and verify the results of manual analysis.

On-chain analysis is the audit of the contracts as they are deployed on the blockchain. This procedure verifies that:

- deployed contracts match those which were audited in manual/static analysis;
- contract values are set to reasonable values;
- contracts are connected so that interdependent contracts function correctly;
- and the ability to modify contract values is restricted via a timelock or DAO mechanism. (We recommend a timelock value of at least 72 hours)

Each obelisk audit is performed by at least two independent auditors who perform their analysis separately.

After the analysis is complete, the auditors will make recommendations for each issue based on best practices and industry standards. The project team can then resolve the issues, and the auditors will verify that the issues have been resolved with no new issues introduced.

Our auditing method lays a particular focus on the following important concepts:

- Quality code and the use of best practices, industry standards, and thoroughly tested libraries.
- Testing the contract from different angles to ensure that it works under a multitude of circumstances.
- Referencing the contracts through databases of common security flaws.

Follow Obelisk Auditing for the Latest Information

ObeliskOrg

ObeliskOrg

OBELISK

Part of Tibereum Group