Cyberscope

Audit Report

Yawn

Aug 2024

Repository https://github.com/cytric-io/cyt-63-yawn-token-smart-contracts

Commit f287a295e71585b3d227093af1c05c63fe102b39

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/cytric-io/cyt-63-yawn-token-smart-contracts
Commit	f287a295e71585b3d227093af1c05c63fe102b39

Audit Updates

Initial Audit	18 Jul 2024 https://github.com/cyberscope-io/audits/blob/main/yawn/v1/audit.pdf
Corrected Phase 2	28 Jul 2024 https://github.com/cyberscope-io/audits/blob/main/yawn/v2/audit.pdf
Corrected Phase 3	05 Aug 2024

Source Files

Filename	SHA256
YawnStaking.sol	bf2ff73dc263103e9554d8af9faf51f8020b0df18806ec19bb4d8e38c055bb47
YawnNFT.sol	09d3d5b12ca9fd00973ef21ee66f45b3a4c0c1097ed9d781d619d7eafb3f9cb7
interfaces/IYawnNFT.sol	7ce1378d4ceff6725b83e77a57d0880f59602a7f7d20d921565419eed9845c14

Overview

YawnNFT contract

The YawnNFT contract is an ERC721 compliant contract designed to manage a series of NFTs (Non-Fungible Tokens) that function as incentives within a staking ecosystem. It includes functionality to mint new NFTs, restricted to the designated staking contract through the `safeMint` function. The contract ensures operational security by incorporating modifiers that restrict access to critical functions.

YawnStaking contract

The YawnStaking contract is designed to manage staking, unstaking, reward distribution, and NFT claiming within a decentralized finance (DeFi) ecosystem. It allows users to stake tokens, earn rewards, and claim NFTs based on their staking activities, while providing functionalities for contract owners to manage important parameters such as the staking token address, distributor address, and locking periods.

Stake Functionality

The stake function allows users to stake a specified amount of tokens into the contract. When a user stakes tokens, the contract insure that the amount of the stake should be greater than the minimumStakedDollarValue and updates the rewards for the user, records the first staking time if it is the user's first stake, and updates the total staked amount. The staked tokens are transferred from the user's address to the contract, and an event is emitted to log the staking action.

Unstake Functionality

The unstake function enables users to withdraw a specified amount of their staked tokens after the nonWithdrawPeriod have passed since their stake. The function ensures the amount to be unstaked is valid and updates the user's rewards accordingly. If the user unstakes tokens before the locking period ends, the rewards for the unstaked amount are distributed to other users. The total staked amount is adjusted, and the tokens are transferred back to the user, with an event emitted to log the unstaking action.

Claim Functionality

The claimRewards function allows users to claim their accumulated rewards. It checks that the user is not within the locking period, updates the user's rewards, transfers the earned tokens to the user, and resets the user's earned rewards. An event is emitted to log the reward claim. Additionally, the claimNFT function allows users to claim an NFT if they meet certain criteria, minting a new NFT for the user and logging the action.

Owner Functionalities

The contract includes several functions for the contract owner to manage key parameters. The owner can set the staking role, set the locking period, and assign a developer wallet address. These functions ensure that the owner can adjust the contract's configuration to adapt to changing requirements or improve the system's security and functionality.

Notify Rewards Functionality

The notifyRewards function allows the distributor to supply tokens to the contract for rewards distribution. It it distributes the rewards and resets the dust. The supplied tokens are transferred from the distributor to the contract, with an event emitted to log the reward distribution.

Reward Calculation and Distribution

The contract includes internal functions to calculate and distribute rewards. The calculateRewards function computes the rewards for a user based on their staked amount and the current reward index. The _updateRewards function updates a user's earned rewards and total earned rewards, adjusting their reward index. The _distributeRewards function distributes the specified amount of rewards to all staked tokens, updating the reward index accordingly.

Initialize frontrun attack

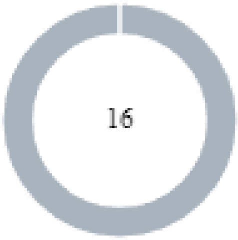
The contracts are vulnerable to an initialize frontrun attack. Deployments can be tracked, allowing an attacker to frontrun and call the `initialize()` function with malicious input before the intended initialization. This issue arises because the contracts do not ensure that the `initialize()` function is called in the same transaction as the wallet deployment, leaving it exposed to potential manipulation by an attacker.

It is recommended to ensure that the `initialize()` function is called within the same transaction as the deployment. This approach will prevent any opportunity for an attacker to frontrun the initialization process and inject malicious input. By securing the initialization process in this manner, the contracts will be protected against frontrun attacks and ensure that the intended initialization parameters are used, maintaining the integrity and security of the deployment process.

Test Deployment

Contract	Explorer
YawnNFT	https://testnet.bscscan.com/address/0xFE9c62422809BC3CCf77516bAEb576077D7acAa2
YawnStaking	https://testnet.bscscan.com/address/0xDa75b2a78505e7992198C4545fef1B2899a253d2

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CCR	Contract Centralization Risk	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	ICC	Inadequate ClaimNFT Criteria	Unresolved
	MN	Misleading Names	Unresolved
	MCP	Missing Constructor Protection	Unresolved
	MEE	Missing Events Emission	Unresolved
	MTDC	Missing Token Decimal Check	Unresolved
	ODM	Oracle Decimal Mismatch	Unresolved
	POSD	Potential Oracle Stale Data	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	RMU	Redundant Modifier Usage	Unresolved
	RC	Repetitive Calculations	Unresolved
	TSI	Tokens Sufficiency Insurance	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L19	Stable Compiler Version	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	YawnNFT.sol#L22 YawnStaking.sol#L183,224,246
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the contract owner has the ability to grant any account the `STAKING_CONTRACT_ROLE` . This role permits the holder to mint tokens to any specified recipient. Consequently, if the owner delegates this role irresponsibly or to malicious actors, it could lead to unchecked minting of tokens, undermining the system's integrity and potentially resulting in the unfair distribution or inflation of tokens. This centralization of authority necessitates robust governance and oversight to mitigate risks.

Additioanlly, the owner has the authority to set critical parameters of the rewards system, which funds the contract and subsequently determines the `rewardIndex` applied to the contract. Any miscalculation or incorrect setting of these parameters can result in incorrect and potentially excessive reward amounts for the users. Additionally, the contract owner has the ability to set the staking token and the locking period, further centralizing control and increasing the risk of mismanagement or exploitation.

```
   function safeMint(address to, uint256 tier) external
onlyRole(STAKING_CONTRACT_ROLE) {
        require(tier > 0, "Invalid tier");

        _safeMint(to, nextId);
        nftTier[nextId] = tier;

        nextId++;
    }
```

```
function notifyRewards(uint256 _reward) external
onlyDistributor(msg.sender) {
    dust += _reward;
    if ((dust * MULTIPLIER) > totalStakedAmount) {
        _distributeRewards(dust);
        dust = 0;
    }
    ERC20(yawn).transferFrom(msg.sender, address(this), _reward);
    emit RewardDistributed(distributor, _reward);
}
function setToken(address _token) external onlyOwner {
    require(_token != address(0), "Invalid address");

    yawn = _token;
}

function setLockingPeriod(uint256 _time) external onlyOwner {
    require(_time > 0, "Invalid time");

    lockingPeriod = _time;
}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	YawnStaking.sol#L287
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
function getMinimumEthAmountNeeded() public view returns (uint256) {
        return (minimumStakedDollarValue * (10 ** 18)) / getLatestPrice();
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

ICC - Inadequate ClaimNFT Criteria

Criticality	Minor / Informative
Location	YawnStaking.sol#L198
Status	Unresolved

Description

The contract is currently configured with a `claimNFT` function that permits users to claim an NFT based solely on the condition of having staked any amount of tokens. This function does not specify a minimum stake requirement, which could potentially lead to exploitation where users stake minimal amounts to mint NFTs, undermining the intended value and rarity of these assets. This loophole might also attract malicious actors who could manipulate the system for personal gain at the expense of genuine stakeholders, thereby destabilizing the ecosystem.

```
function claimNFT() external {
        require(isClaimableNFT(msg.sender), "Not claimable a NFT");
        stakes[msg.sender].claimedTimeForNFT = block.timestamp;
        yawnNFT.safeMint(msg.sender, nftTier);
        emit NFTClaimed(msg.sender, yawnNFT.nextId() - 1);
    }
```

Recommendation

It is recommended to introduce a minimum cap on the staking amount required for users to claim NFTs. This adjustment will ensure that only participants who contribute substantially to the ecosystem can access the benefits of NFT minting, thereby aligning user incentives with the long-term health and value of the platform. Implementing this cap will also help maintain the exclusivity and intrinsic value of the minted NFTs, fostering a more balanced and sustainable environment.

MN - Misleading Names

Criticality	Minor / Informative
Location	YawnStaking.sol#L216
Status	Unresolved

Description

The contract is currently designed with an internal function named `_distributeRewards` and a variable named `rewardIndex`. However, both the function name and the variable name are misleading. The `_distributeRewards` function does not actually distribute any rewards but rather updates the `rewardIndex`, which itself represents token allocation rather than an actual reward index. This misnaming can cause confusion, potentially leading to misunderstandings about the contract's functionality and logic.

```
function _distributeRewards(uint256 _amount) internal {
        require(totalStakedAmount > 0, "No staked token");

        rewardIndex += (_amount * MULTIPLIER) / totalStakedAmount;
    }
```

Recommendation

It is recommended to reconsider the naming of the function and variable to more accurately reflect their purposes. The function could be renamed to clarify that the function updates the token allocation index rather than distributing rewards. This change will improve the readability and maintainability of the contract, ensuring that its functionality is clear to all stakeholders.

MCP - Missing Constructor Protection

Criticality	Minor / Informative
Location	YawnNFT.sol#L14 YawnStaking.sol#L56
Status	Unresolved

Description

The contract is missing the `_disableInitializers()` function in its constructor. This function was introduced in Contracts v4.6.0 to support reinitializers and provides an essential safeguard by preventing the initialization of the implementation contract. Without this protection, there is a risk that an attacker could initialize the contract, potentially leading to a malicious takeover of the implementation contract. This vulnerability arises because the constructor does not currently include the `_disableInitializers()` call, leaving the contract exposed to initialization attacks.

```
constructor() {}
```

Recommendation

It is recommended to include the `_disableInitializers()` function within the constructor to prevent unauthorized initialization of the implementation contract. This additional protection will mitigate the risk of an attacker gaining control by initializing the contract. By incorporating this safeguard, the contract will adhere to the updated security practices introduced in Contracts v4.6.0 and ensure greater robustness against potential attacks.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	YawnStaking.sol#L81,246,284
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
minimumStakedDollarValue = _minimumStakedDollarValue
lockingPeriod = _time;
minimumStakedDollarValue = _value;
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MTDC - Missing Token Decimal Check

Criticality	Minor / Informative
Location	YawnStaking.sol#L224
Status	Unresolved

Description

The contract contains the `setToken` function, which allows the owner to set the token address used by the contract. However, the function does not include a `require` check to verify that the token has 18 decimals, as is enforced in the contract's constructor. This omission could lead to inconsistencies and potential issues if a token with a different decimal configuration is set, affecting calculations and the overall functionality of the contract.

```
function setToken(address _token) external onlyOwner {
        require(_token != address(0), "Invalid address");

        yawn = _token;
    }
```

Recommendation

It is recommended to include a `require` check within the `setToken` function to verify that the token's decimals are equal to 18. This will ensure consistency with the initial contract setup and prevent potential issues arising from tokens with incompatible decimal configurations. Adding this check will enhance the contract's robustness and reliability.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	YawnStaking.sol#L277
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
function getLatestPrice() public view returns (uint256) {
        (, int256 price, , , ) = priceFeed.latestRoundData();
        price = (price * (10 ** 10));
        return uint256(price);
    }
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	YawnStaking.sol#L277
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
function getLatestPrice() public view returns (uint256) {
        (, int256 price, , , ) = priceFeed.latestRoundData();
        price = (price * (10 ** 10));
        return uint256(price);
    }
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	YawnStaking.sol#L114,116
Status	Unresolved

Description

The `safeTransferFrom` function is used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
totalStakedAmount += _amount;

ERC20(yawn).safeTransferFrom(msg.sender, address(this), _amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

RMU - Redundant Modifier Usage

Criticality	Minor / Informative
Location	YawnStaking.sol#L85,113
Status	Unresolved

Description

The contract contains the `onlyDistributor` modifier, which is used to restrict access to the `notifyRewards` function. However, this is the only function where the `onlyDistributor` modifier is applied. Given its limited use, the inclusion of this modifier adds unnecessary complexity and gas costs to the contract. Instead, a simple `require` statement within the `notifyRewards` function could achieve the same access control, making the code more efficient and easier to maintain.

```
modifier onlyDistributor(address user) {
    require(user == distributor, "Invalid distributor");
    _;
}

function notifyRewards(uint256 _reward) external
onlyDistributor(msg.sender) {
    dust += _reward;
    if ((dust * MULTIPLIER) > totalStakedAmount) {
        _distributeRewards(dust);
        dust = 0;
    }
    ERC20(yawn).transferFrom(msg.sender, address(this), _reward);
    emit RewardDistributed(distributor, _reward);
}
```

Recommendation

It is recommended to replace the `onlyDistributor` modifier with a `require` statement directly within the `notifyRewards` function. This change will reduce gas costs and simplify the contract without compromising security or functionality. By streamlining the access control mechanism, the contract will become more efficient and maintainable.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	YawnStaking.sol#L208,209
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
earned[_user] += calculateRewards(_user);
totalEarned[_user] += calculateRewards(_user);
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	YawnStaking.sol#L183
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

Specifically, the distributor address has the authority to invoke the `notifyRewards` function to activate the rewards for users by defining the correct amount of tokens to be distributed. This dependency on a single distributor could lead to potential misuse or delays in reward distribution, further centralizing control and creating a single point of failure.

```
function notifyRewards(uint256 _reward) external
onlyDistributor(msg.sender) {
    dust += _reward;
    if ((dust * MULTIPLIER) > totalStakedAmount) {
        _distributeRewards(dust);
        dust = 0;
    }
    ERC20(yawn).transferFrom(msg.sender, address(this), _reward);
    emit RewardDistributed(distributor, _reward);
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	YawnStaking.sol#L59,60,61,62,63,64,93,124,183,193,224,233,243,249,254,259,263,267,283
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _token
address _nft
address _oracle
address _router
uint256 _minimumStakedDollarValue
uint256 _nftTier
uint256 _amount
uint256 _reward
address _user
address _distributor
uint256 _time
address _wallet
uint256 _value
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	YawnStaking.sol#L73,237
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
distributor = msg.sender
distributor = _distributor
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	YawnStaking.sol#L2 YawnNFT.sol#L2 interfaces/IYawnNFT.sol#L3
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

Recommendation

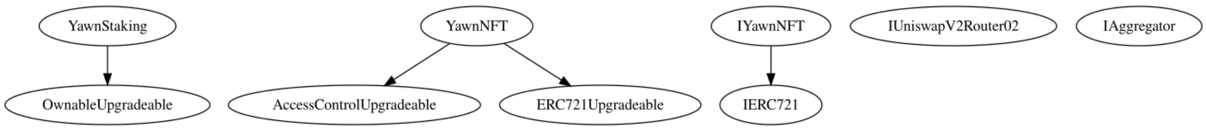
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

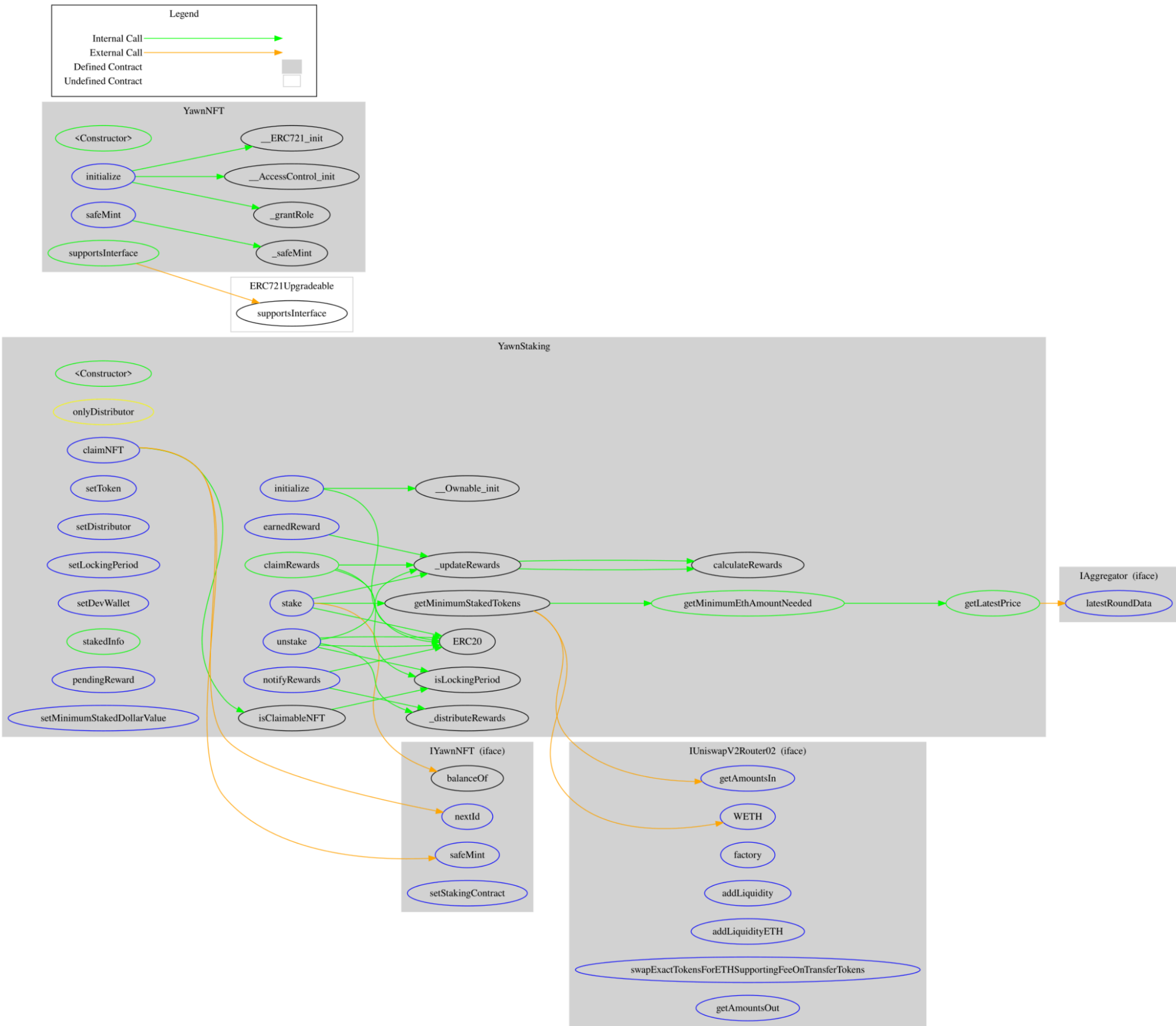
Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
YawnStaking	Implementation	OwnableUpgradeable		
		Public	✓	-
	initialize	External	✓	initializer
	stake	External	✓	-
	unstake	External	✓	-
	claimRewards	Public	✓	-
	notifyRewards	External	✓	onlyDistributor
	calculateRewards	Public		-
	claimNFT	External	✓	-
	_updateRewards	Internal	✓	
	_distributeRewards	Internal	✓	
	setToken	External	✓	onlyOwner
	setDistributor	External	✓	onlyOwner
	setLockingPeriod	External	✓	onlyOwner
	setDevWallet	External	✓	onlyOwner
	isLockingPeriod	Public		-
	isClaimableNFT	Public		-
	stakedInfo	Public		-
	earnedReward	External	✓	-
	pendingReward	External		-

	getLatestPrice	Public		-
	setMinimumStakedDollarValue	External	✓	onlyOwner
	getMinimumEthAmountNeeded	Public		-
	getMinimumStakedTokens	Public		-
YawnNFT	Implementation	AccessControlUpgradeable, ERC721Upgradeable		
		Public	✓	-
	initialize	External	✓	initializer
	safeMint	External	✓	onlyRole
	supportsInterface	Public		-
IYawnNFT	Interface	IERC721		
	safeMint	External	✓	-
	setStakingContract	External	✓	-
	nextId	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
YawnNFT
<Constructor>
__ERC721_init
initialize
__AccessControl_init
safeMint
_grantRole
supportsInterface
_safeMint
ERC721Upgradeable
supportsInterface
YawnStaking
<Constructor>
onlyDistributor
claimNFT
setToken
initialize
__Ownable_init
setDistributor
earnedReward
setLockingPeriod
claimRewards
_updateRewards
calculateRewards
setDevWallet
stake
getMinimumStakedTokens
getMinimumEthAmountNeeded
getLatestPrice
stakedInfo
unstake
ERC20
pendingReward
notifyRewards
isLockingPeriod
setMinimumStakedDollarValue
isClaimableNFT
_distributeRewards
IAggregator (iface)
latestRoundData
IYawnNFT (iface)
balanceOf
nextId
safeMint
setStakingContract
IUniswapV2Router02 (iface)
getAmountsIn
WETH
factory
addLiquidity
addLiquidityETH
swapExactTokensForETHSupportingFeeOnTransferTokens
getAmountsOut

Summary

The Yawn project is composed of the YawnNFT contract, which implements an ERC721-compliant NFT mechanism for minting and managing NFTs within a staking ecosystem, and the YawnStaking contract, which handles staking, unstaking, reward distribution, and NFT claiming processes. This audit investigates security issues, business logic concerns, and potential improvements across both contracts, focusing on minting restrictions, operator permissions, staking validation, reward calculation, and centralized control risks to ensure fair, secure, and efficient operation.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io