SoluLab

Security Assessment
(Final Report)

PTEK Smart Contracts

NFTCreationContract, NFTStakingContract and NFTStakingRewardWallet

September 27th, 2024

www.solulab.com

Table of Contents

Summary

This report presents a comprehensive audit of the NFTCreationContract, NFTStakingRewardWallet, and NFTStakingContract smart contracts. These contracts are designed to work together in managing NFT minting, staking rewards, and PTEK token locking within the platform. The audit's primary goal was to identify potential vulnerabilities, verify the integrity of the contract logic, and ensure compliance with best practices and project requirements.

The auditing process involved:
- Analyzing the smart contracts for vulnerability to common and advanced attack vectors.
- Reviewing the codebase for adherence to current best practices and industry standards.
- Verifying that the contract logic fulfills the project's specifications and goals.

The primary areas of focus during the audit included:
- Reentrancy prevention, especially in functions handling fund transfers.
- Correct implementation of reward distribution based on global locking stages.
- Gas optimization to improve overall contract efficiency.
- Correct handling of access control through the onlyOwner modifier and other mechanisms.
- Proper validation and state changes during NFT minting, PTEK locking, and reward distribution.

The security assessment revealed issues ranging from minor to critical in severity. Critical issues require immediate attention to prevent potential security breaches, while minor issues are mostly recommendations for improving code efficiency and maintainability.

Recommendations from this audit include:
- Implementing reentrancy guards to protect against potential attacks.
- Optimizing storage patterns to reduce gas consumption.
- Enhancing event emission practices for improved transparency and monitoring.

-Ensure reward distribution logic accurately calculate rewards based on user stake, PTEK rate, and stage-specific allowance unlocking percentage, ensuring fairness and consistency across all stages
-If there is no specific requirement for using Solidity version 0.8.19, it is advisable to upgrade to version 0.8.24. This version offers the same stability with added improvements and optimizations.
-Follow the Solidity naming convention.

Some of the above recommendations have not yet been implemented.
Please ensure that all recommendations are followed as closely as possible to improve code security, performance, and readability.

Addressing the identified issues and recommendations will significantly enhance the security posture and operational efficiency of the NFTCreationContract, NFTStakingRewardWallet, and NFTStakingContract contracts. Adhering to these suggestions will ensure the contracts meet high-security standards and align with industry best practices.

Overview

Project Summary:

Project Name	PTEK Smart Contract
Description	The Audit report of smart contracts with Static and Manual analysis.
Platform	EVM Compatible Chains
Smart Contract Language	Solidity
Codebase	PTEK Smart Contract
Contract Address	NFTCreationContract NFTStakingContract NFTStakingRewardWallet

Audit Summary

Delivery Date	TBD
Audit Methodology	Static analysis, Manual Review, Functional Testing

Vulnerability Summary

	Initial Audit	Final Audit Status
Total Issues	23	No new issues
Low	14	Fixed
Moderate	2	Partially Fixed*
High	7	Fixed

* Follow recommendations for improved code security

Findings

ID	File Name	Category	Severity	Status
NFTCreationContract	NFTCreationContract.sol	Gas Optimization, Coding style	Low, moderate	Low -Fixed Moderate - Partially Fixed*
NFTStakingRewardWallet	NFTStakingRewardWallet.sol	Gas Optimization, Coding style	Low, moderate	Low -Fixed Moderate - Partially Fixed*

* Follow recommendations for improved code security

1) Reentrancy (moderate)

Issue- The current use of the onlyOwner modifier in functions **send** and **safeMint** helps mitigate reentrancy attack risks by restricting access to the contract's owner.Even with access control, it's essential to architect smart contracts with inherent protections against reentrancy attacks.

```
Reentrancy in NFTStakingRewardWallet.send(address,uint256) (contracts/NFTStakingRewardWallet.sol#31-43):
```

```
Reentrancy in NFTCreationContract.safeMint(address,string) (contracts/NFTCreationContract.sol#29-35):
```

Recommendation- Consider using a reentrancy guard modifier for functions making external calls or transferring Ether. This practice ensures that the contract remains secure against reentrancy attacks.Also apply the checks-effects-interactions pattern by ensuring that internal state changes occur before making any external calls.

Status- Partially fixed by the developer. In the NFTCreationContract.safeMint(address,string), the state-changing operation (_setTokenURI) is executed after the external call (_safeMint). Similarly, in the NFTStakingRewardWallet.send(address,uint256) function, the external call is made before emitting the NFTRewardSent event.

2) Low level call(Low)

Issue- Low-level calls are error-prone as they do not check for the existence of the called contract's code or whether the call succeeded, leading to potential errors or silent failures.

```
Low level call in NFTStakingRewardWallet.send(address,uint256) (contracts/NFTStakingRewardWallet.sol#31-43):
        - (sent) = to.call{value: amount}(abi.encodeWithSignature(1)) (contracts/NFTStakingRewardWallet.sol#36)
```

Recommendation- Avoid using low-level calls unless absolutely necessary. Prefer using higher-level functions like transfer, send etc, as they offer better safety, automatic error handling, and clarity.

Status- PropTech requires the send function to include both value and data in the transaction, which is implemented using a low-level call mechanism.

3) Enhanced Documentation (Low)

Issue- The current smart contract lacks NatSpec comments, missing an opportunity to provide valuable documentation directly within the code.

Recommendation- Implement NatSpec comments throughout the contract. This includes descriptive summaries for contract functionality, parameters, return values, and any notable side effects or requirements.

Status - Fixed by the developer.

4) Contract Flexibility with Upgradeable Functions (Low)

Issue- The current implementation of the NFTStakingContract and NFTStakingRewardWallet contracts does not utilize upgradeable patterns. This limitation restricts the ability to address bugs, improve functionality, or adapt to new requirements post-deployment.

Recommendation- Implement upgradeable contracts using a proxy pattern, such as those provided by established frameworks like OpenZeppelin's Upgrades. This approach allows for the modification and enhancement of contract logic without losing the existing state or redeploying the contract.

Status - PropTech chose not to implement a flexible and upgradeable strategy because they aimed to create an immutable smart contract. This approach ensures that the contract cannot be changed, offering greater confidence and protection to users.

Slither Tool Result:-

1. NFTStakingRewardWallet

```
Reentrancy in NFTStakingRewardWallet.send(address,uint256) (contracts/NFTStakingRewardWallet.sol#57-64):
        External calls:
        - (sent) = to.call{value: amount}(abi.encodeWithSignature(1)) (contracts/NFTStakingRewardWallet.sol#60)
        Event emitted after the call(s):
        - NFTRewardSent(to,amount) (contracts/NFTStakingRewardWallet.sol#62)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Pragma version=0.8.19 (contracts/NFTStakingRewardWallet.sol#3) necessitates a version too recent to be trusted. Consider de
ploying with 0.8.18.
solc-0.8.19 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in NFTStakingRewardWallet.send(address,uint256) (contracts/NFTStakingRewardWallet.sol#57-64):
        - (sent) = to.call{value: amount}(abi.encodeWithSignature(1)) (contracts/NFTStakingRewardWallet.sol#60)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Modifier NFTStakingRewardWallet.ReentrancyGuard() (contracts/NFTStakingRewardWallet.sol#31-36) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

2. NFTCreationContract

```
Reentrancy in NFTCreationContract.safeMint(address,string) (contracts/NFTCreationContract.sol#50-56):
        External calls:
        - _safeMint(to,tokenId) (contracts/NFTCreationContract.sol#53)
                - retval = IERC721Receiver(to).onERC721Received(_msgSender(),from,tokenId,data) (node_modules/@openzeppelin/contracts/token/ERC72
1/ERC721.sol#401-412)
        State variables written after the call(s):
        - _setTokenURI(tokenId,uri) (contracts/NFTCreationContract.sol#54)
                - _tokenURIs[tokenId] = _tokenURI (node_modules/@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
```

```
INFO:Detectors:
Modifier NFTCreationContract.ReentrancyGuard() (contracts/NFTCreationContract.sol#33-38) is not in mixedCase
```

ID	File Name	Category	Severity	Status
NFTStakingContract	NFTStakingContract.sol	Gas Optimization, Coding style, Arithmetic operations,missing logic	Low, moderate, high	High, Low - Fixed Moderate - Partially Fixed*

* Follow recommendations for improved code security

1) Missing Semicolon [Compilation Error] (high)

Issue-The safeMint function and dailyReward in the NFTStakingContract is missing a semicolon at the end of a statement, which is causing a compilation error.

```
1  users[_user].totalLocked =  rewardPTEK + users[_user].packagePTEK;
2  users[_user].totalStaked = users[_user].totalLocked
3
4  users[_user].exchangeRate =  _exchangeRate;
5
```

```
ParserError: Expected ';' but got identifier
   --> contracts/NFTStakingContract.sol:134:9:
    |
134 |          users[_user].exchangeRate =  _exchangeRate;
    |          ^^^^^

Error HH600: Compilation failed
```

```
1  require(users[user].lastPaid + 1 days >= dailyLockTime, "error: daily reward already unlocked for today");
2
3       rewardPercentage = users[user].package.dailyReward
4       if (currentPTEKRate > users[user].exchangeRate) {
```

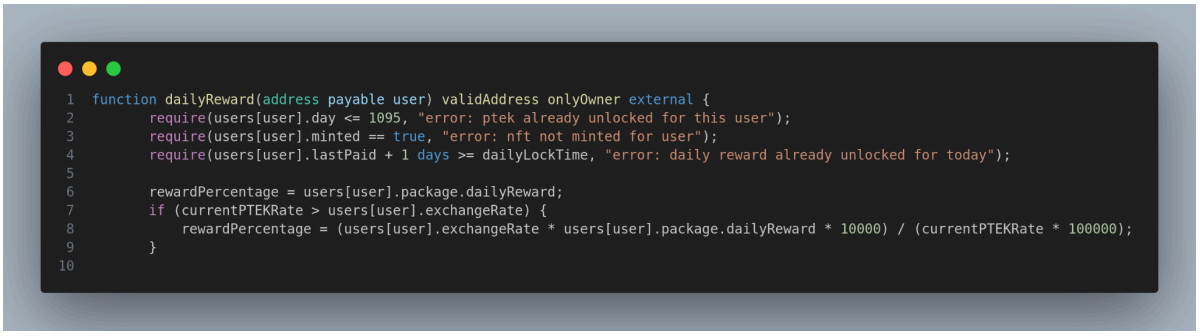
```
ParserError: Expected ';' but got 'if'
   --> contracts/NFTStakingContract.sol:205:9:
    |
205 |          if (currentPTEKRate > users[user].exchangeRate) {
    |          ^^
```

Recommendation- Include the semicolon to fix this compilation error.

Status - Fixed by the developer.

2)Undeclared Identifier [Compilation Error] (high)

Issue-The identifier rewardPercentage is used in the contract code but has not been declared. This undeclared identifier results in a compilation error and prevents the contract from functioning as intended.



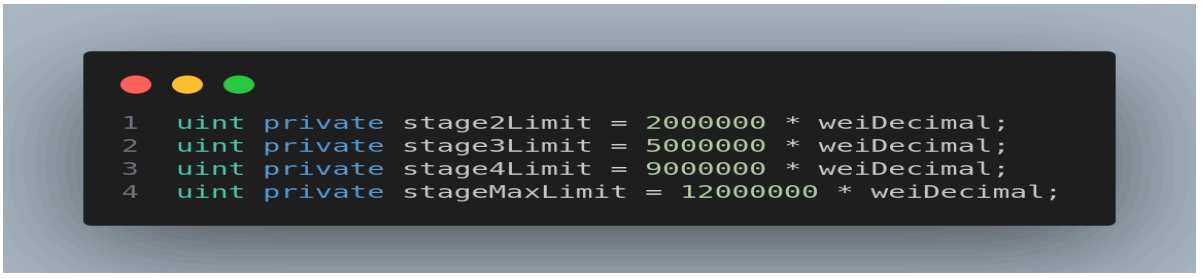
```
function dailyReward(address payable user) validAddress onlyOwner external {
        require(users[user].day <= 1095, "error: ptek already unlocked for this user");
        require(users[user].minted == true, "error: nft not minted for user");
        require(users[user].lastPaid + 1 days >= dailyLockTime, "error: daily reward already unlocked for today");

        rewardPercentage = users[user].package.dailyReward;
        if (currentPTEKRate > users[user].exchangeRate) {
            rewardPercentage = (users[user].exchangeRate * users[user].package.dailyReward * 10000) / (currentPTEKRate * 100000);
        }

```

Recommendation- Declare the rewardPercentage variable before using it in the contract. Ensure it has the correct data type and scope to be used within the function or contract where needed.

Status - Fixed by the developer.

3) State-variables (Low)

Issue- These stage limits are not updated following deployment, yet it is not declared as immutable.



```
uint private stage2Limit = 2000000 * weiDecimal;
uint private stage3Limit = 5000000 * weiDecimal;
uint private stage4Limit = 9000000 * weiDecimal;
uint private stageMaxLimit = 12000000 * weiDecimal;
```

```
NFTStakingContract.stage2Limit (contracts/NFTStakingContract.sol#54) should be immutable
NFTStakingContract.stage3Limit (contracts/NFTStakingContract.sol#55) should be immutable
NFTStakingContract.stage4Limit (contracts/NFTStakingContract.sol#56) should be immutable
NFTStakingContract.stageMaxLimit (contracts/NFTStakingContract.sol#57) should be immutable
```

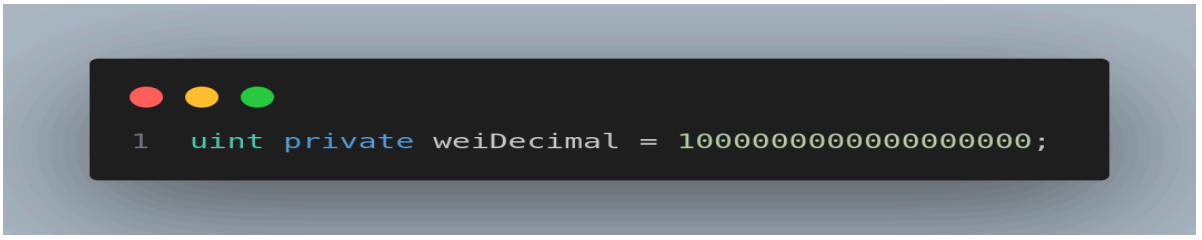
Recommendation- Declare these stage limit variables as immutable.By declaring the variables as **immutable**, accessing these stage limits will be cheaper compared to regular state variables, which would involve more expensive storage reads. This results in gas savings across any function calls that interact with these limits.

Status - Fixed by the developer.

4) Declaration of weiDecimal(Low)

Issue- The weiDecimal variables are used to define no. of wei in a PTEK but are not declared as constants. These values stay the same after the contract is deployed, and their current declaration may lead to unnecessary gas costs when they are accessed.

```
NFTStakingContract.weiDecimal (contracts/NFTStakingContract.sol#52) should be constant
```



Recommendation- Declare weiDecimal as constant. This can be done by adding the constant keyword to their declarations.

Status - Fixed by the developer.

5) Uses literals with too many digit(Low)

Issue- The weiDecimal Literals with many digits are difficult to read and review.

```
NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#199-223) uses literals with too many digits:
        - rewardPercentage = (users[user].exchangeRate * users[user].package.dailyReward * 10000) / (currentPTEKRate * 100000) (contracts/NFTStakingContract.sol#206)
NFTStakingContract.slitherConstructorVariables() (contracts/NFTStakingContract.sol#8-228) uses literals with too many digits:
        - weiDecimal = 1000000000000000000 (contracts/NFTStakingContract.sol#52)
NFTStakingContract.slitherConstructorVariables() (contracts/NFTStakingContract.sol#8-228) uses literals with too many digits:
        - stage2Limit = 2000000 * weiDecimal (contracts/NFTStakingContract.sol#54)
NFTStakingContract.slitherConstructorVariables() (contracts/NFTStakingContract.sol#8-228) uses literals with too many digits:
        - stage3Limit = 5000000 * weiDecimal (contracts/NFTStakingContract.sol#55)
NFTStakingContract.slitherConstructorVariables() (contracts/NFTStakingContract.sol#8-228) uses literals with too many digits:
        - stage4Limit = 9000000 * weiDecimal (contracts/NFTStakingContract.sol#56)
NFTStakingContract.slitherConstructorVariables() (contracts/NFTStakingContract.sol#8-228) uses literals with too many digits:
        - stageMaxLimit = 12000000 * weiDecimal (contracts/NFTStakingContract.sol#57)
```

```
1  uint private weiDecimal = 1000000000000000000;
```

Recommendation- Use Ether Suffix to make the code more readable.

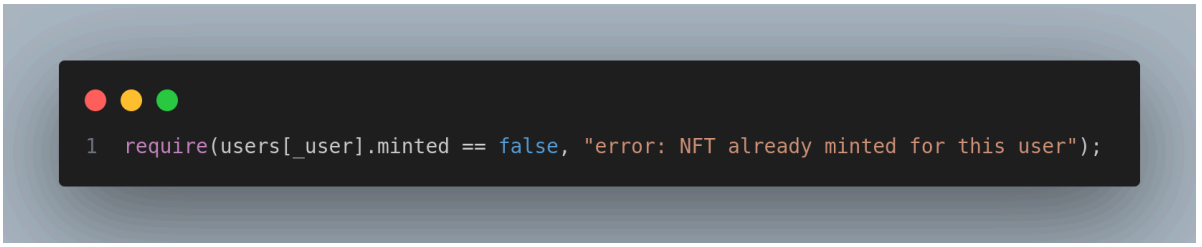
Status- Proptech agreed that the use of literals with excessive digits is a concern; however, they are not making alterations at this time to avoid extending the testing cycle related to decimal precision.

6) Boolean comparison with constant values(Low)

Issue- In the current implementation of NFTStakingContract, several functions compare boolean variables to constant values (true or false) in require statements.

```
NFTStakingContract.initUser(address,uint256) (contracts/NFTStakingContract.sol#106-113) compares to a boolean constant:
        -require(bool,string)(users[_user].active == false,error: user already exists) (contracts/NFTStakingContract.sol#108)
NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#115-147) compares to a boolean constant:
        -require(bool,string)(users[_user].minted == false,error: NFT already minted for this user) (contracts/NFTStakingContract.sol#116)
NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#199-223) compares to a boolean constant:
        -require(bool,string)(users[user].minted == true,error: nft not minted for user) (contracts/NFTStakingContract.sol#201)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
```

```
1  require(users[_user].active == false, "error: user already exists");
```



Recommendation- By removing the redundant comparison to boolean constants, you reduce the complexity of your code, making it cleaner and more gas-efficient.

Status - Fixed by the developer.

7) Event Emission for State Changes (Low)

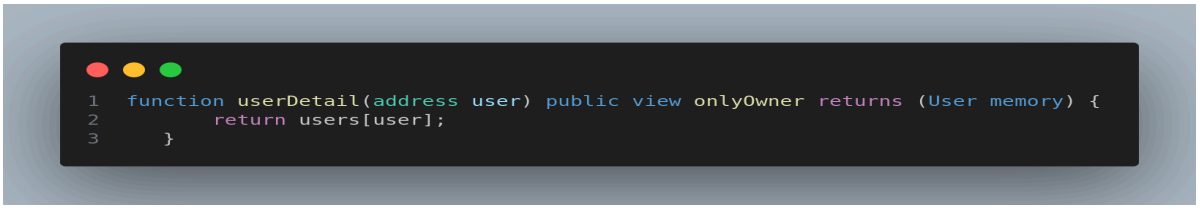
Issue- The functions (initUser, mintNFT, updateStage, dailyLocking, dailyReward) of NFTStakingContract that alter the contract's state but do not emit events following these changes. Emitting events after state changes is a best practice in smart contract development.

Recommendation- Introduce new events that correspond to the actions performed within the above mentioned functions. Emit these events immediately after the state changes occur within these functions.

Status - Fixed by the developer.

8) Public Visibility for Functions (Low)

Issue- The userDetail function is currently marked as public, implying they can be called both internally and externally.



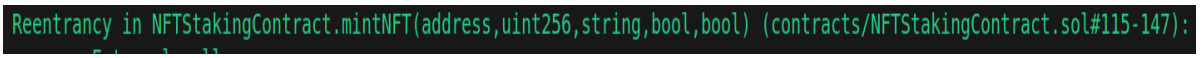
```
function userDetail(address user) public view onlyOwner returns (User memory) {
        return users[user];
    }
```

Recommendation- Change the visibility of the functions from public to external. The external visibility specifier is more gas-efficient for functions that are intended to be called only from outside the contract. This change not only optimizes gas usage but also clarifies the intended use of these functions.

Status - Fixed by the developer.

9) Reentrancy (moderate)

Issue-The current use of the onlyOwner modifier in functions mintNFT and dailyReward helps mitigate reentrancy attack risks by restricting access to the contract's owner. Even with access control, it's essential to architect smart contracts with inherent protections against reentrancy attacks.



```
Reentrancy in NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#115-147):
```

Recommendation- Consider using a reentrancy guard modifier for functions making external calls or transferring Ether. This practice ensures that the contract remains secure against reentrancy attacks.Also apply the checks-effects-interactions pattern by ensuring that internal state changes occur before making any external calls.

Status- Partially fixed by the developer. In the NFTStakingContract, several functions involve external calls being made before state-changing operations.

In lockReward(address,uint256), the external call (nftRewardWallet.send) is executed before updating the stage

(updateStage(currentStage + 1)), aligning with business logic to distribute rewards first and then update the stage.

In mintNFT(address,uint,string,bool,bool), the external call to mint the NFT (soulBoundNFT.safeMint) happens before updating the user's information (users[user] = _user).

Similarly, in dailyReward(address), the reward is sent to the user (user.send(dailyRewardPTEK)) before updating the user's reward status (users[user] = _user).

In the payPendingReward(address,uint) function of the NFTStakingContract, the external call to transfer dailyRewardPTEK to the user occurs before updating the user's information and emitting the PendingRewardUnlocked event.

10) Storage Access in Contract Functions (Low)

Issue- The functions dailyReward currently perform multiple reads from storage for the same variables, leading to increased gas costs and less efficient execution.

Recommendation- Optimize this function by minimizing redundant storage reads. Read the necessary data from storage once at the beginning of the function, store it in local variables, and use these local variables throughout the function. This approach reduces gas consumption and enhances the function's efficiency.

Status - Fixed by the developer.

11) Enhanced Documentation (Low)

Issue- The current smart contract lacks NatSpec comments, missing an opportunity to provide valuable documentation directly within the code.

Recommendation- Implement NatSpec comments throughout the contract. This includes descriptive summaries for contract functionality, parameters, return values, and any notable side effects or requirements.

Status - Fixed by the developer.

12) Block Timestamp for Control Decisions (Low)

Issue- The dailyLocking function relies on block.timestamp to determine whether the PTEK rate and global PTEK locked values can be updated. Using block.timestamp introduces potential risk, as miners can slightly manipulate it to alter the behavior of the contract.

```
function dailyLocking(uint _PTEKRate, uint _globalPTEKLocked) onlyOwner external {
    // set rate and glocal ptek locked at 00:01 AM for paying today
    require(block.timestamp >= dailyLockTime, "error: wait atleast 24 hours for next update!");
```

Recommendation- Include a time buffer to mitigate the risk of miner manipulation. (For example 1 hour)

Status-PropTech has determined that adding any buffer time to block.timestamp compromises the integrity of the check and disrupts the functionality, making the time-sensitive logic ineffective.As The block.timestamp must strictly be 12:01 AM for the intended functionality to work as designed.

13) Contract Interactions with Interfaces (Low)

Issue- The direct deployment of NFTCreationContract and NFTStakingRewardWallet from the NFTStakingContract constructor, without the use of interfaces, limits the system's modularity and flexibility.

Recommendation-Leverage Solidity interfaces to define how NFTStakingContract interacts with NFTStakingRewardWallet and NFTCreationContract. Deploy these contracts independently and set their addresses in NFTStakingContract post-deployment. This decouples

the contracts, allowing each to be updated or replaced without impacting the others. Using interfaces enhances contract modularity and can lead to more efficient gas usage during interactions.

Status - PropTech chooses not to use separate deployments for NFTStakingRewardWallet and NFTCreationContract for their specific use cases.

14) Contract Flexibility with Upgradeable Functions (Low)

Issue- The current implementation of the NFTStakingContract contracts does not utilize upgradeable patterns. This limitation restricts the ability to address bugs, improve functionality, or adapt to new requirements post-deployment.

Recommendation- Implement upgradeable contracts using a proxy pattern, such as those provided by established frameworks like OpenZeppelin's Upgrades. This approach allows for the modification and enhancement of contract logic without losing the existing state or redeploying the contract.

Status - PropTech chose not to implement a flexible and upgradeable strategy because they aimed to create an immutable smart contract. This approach ensures that the contract cannot be changed, offering greater confidence and protection to users.

Slither Tool Result:-

```
Reentrancy in NFTStakingContract.payPendingReward(address,uint256) (contracts/NFTStakingContract.sol#219-248):
        External calls:
        - success = user.send(dailyRewardPTEK) (contracts/NFTStakingContract.sol#232)
        State variables written after the call(s):
        - users[user] = _user (contracts/NFTStakingContract.sol#245)
        Event emitted after the call(s):
        - PendingRewardUnlocked(user,dailyRewardPTEK,block.timestamp) (contracts/NFTStakingContract.sol#247)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-4
```

```
Constant NFTStakingContract.weiDecimal (contracts/NFTStakingContract.sol#47) is not in UPPER_CASE_WITH_UNDERSCORES
Modifier NFTStakingContract.ReentrancyGuard() (contracts/NFTStakingContract.sol#131-136) is not in mixedCase
Modifier NFTStakingRewardWallet.ReentrancyGuard() (contracts/NFTStakingRewardWallet.sol#31-36) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Reentrancy in NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#293-330):
       External calls:
       - success = user.send(dailyRewardPTEK) (contracts/NFTStakingContract.sol#320)
       State variables written after the call(s):
       - users[user] = _user (contracts/NFTStakingContract.sol#327)
       Event emitted after the call(s):
       - DailyRewardUnlocked(user,dailyRewardPTEK,block.timestamp) (contracts/NFTStakingContract.sol#329)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-4
INFO:Detectors:
Variable NFTStakingContract.stage2Limit (contracts/NFTStakingContract.sol#50) is too similar to NFTStakingContract.stage3Limit (contracts/NFTStak
ingContract.sol#51)
Variable NFTStakingContract.stage2Limit (contracts/NFTStakingContract.sol#50) is too similar to NFTStakingContract.stage4Limit (contracts/NFTStak
ingContract.sol#52)
Variable NFTStakingContract.stage3Limit (contracts/NFTStakingContract.sol#51) is too similar to NFTStakingContract.stage4Limit (contracts/NFTStak
ingContract.sol#52)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar
INFO:Detectors:
NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#293-330) uses literals with too many digits:
       - rewardPercentage = (_user.exchangeRate * _user.package.dailyReward * 10000) / (currentPTEKRate * 100000) (contracts/NFTStakingContract.
sol#303)
NFTStakingContract.slitherConstructorConstantVariables() (contracts/NFTStakingContract.sol#14-345) uses literals with too many digits:
       - weiDecimal = 1000000000000000000 (contracts/NFTStakingContract.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
```

```
INFO:Detectors:
Reentrancy in NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229):
        External calls:
        - success = nftRewardWallet.send(address(address(this)),rewardPTEK) (contracts/NFTStakingContract.sol#217)
        Event emitted after the call(s):
        - StageUpdated(currentStage,block.timestamp) (contracts/NFTStakingContract.sol#269)
                - updateStage(currentStage + 1) (contracts/NFTStakingContract.sol#226)
        - UserRewardLocked(user,rewardPTEK,block.timestamp) (contracts/NFTStakingContract.sol#228)
Reentrancy in NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#195-2
06):
        External calls:
        - lockReward(user,exchangeRate) (contracts/NFTStakingContract.sol#198)
                - success = nftRewardWallet.send(address(address(this)),rewardPTEK) (contracts/NFTStakingContract.
sol#217)
        - _tokenId = soulBoundNFT.safeMint(user,uri) (contracts/NFTStakingContract.sol#200)
        Event emitted after the call(s):
        - NFTMinted(user, _tokenId,block.timestamp) (contracts/NFTStakingContract.sol#205)
```

```
INFO:Detectors:
NFTStakingContract.dailyLocking(uint256,uint256) (contracts/NFTStakingContract.sol#282-288) uses timestamp for com
parisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= dailyLockTime,error: wait atleast 24 hours for next update!) (co
ntracts/NFTStakingContract.sol#283)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

```
INFO:Detectors:
NFTStakingContract.dailyReward(address).user (contracts/NFTStakingContract.sol#293) lacks a zero-check on :
                - success = user.send(dailyRewardPTEK) (contracts/NFTStakingContract.sol#320)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229):
        External calls:
        - success = nftRewardWallet.send(address(address(this)),rewardPTEK) (contracts/NFTStakingContract.sol#217)
        State variables written after the call(s):
        - updateStage(currentStage + 1) (contracts/NFTStakingContract.sol#226)
                - currentStage = stage (contracts/NFTStakingContract.sol#244)
                - currentStage = stage (contracts/NFTStakingContract.sol#256)
                - currentStage = stage (contracts/NFTStakingContract.sol#267)
        - updateStage(currentStage + 1) (contracts/NFTStakingContract.sol#226)
                - nextStageTrigger = stage3Limit (contracts/NFTStakingContract.sol#243)
                - nextStageTrigger = stage4Limit (contracts/NFTStakingContract.sol#255)
        - totalPtekLocked += rewardPTEK (contracts/NFTStakingContract.sol#219)
```

```
INFO:Detectors:
Reentrancy in NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229):
        External calls:
        - success = nftRewardWallet.send(address(address(this)),rewardPTEK) (contracts/NFTStakingContract.sol#217)
        State variables written after the call(s):
        - updateStage(currentStage + 1) (contracts/NFTStakingContract.sol#226)
                - packages[50] = Package(225,25) (contracts/NFTStakingContract.sol#237)
                - packages[100] = Package(225,25) (contracts/NFTStakingContract.sol#238)
                - packages[500] = Package(250,30) (contracts/NFTStakingContract.sol#239)
                - packages[1500] = Package(250,30) (contracts/NFTStakingContract.sol#240)
                - packages[5000] = Package(275,35) (contracts/NFTStakingContract.sol#241)
                - packages[8000] = Package(275,35) (contracts/NFTStakingContract.sol#242)
                - packages[50] = Package(200,20) (contracts/NFTStakingContract.sol#249)
                - packages[100] = Package(225,25) (contracts/NFTStakingContract.sol#250)
                - packages[500] = Package(225,25) (contracts/NFTStakingContract.sol#251)
                - packages[1500] = Package(250,30) (contracts/NFTStakingContract.sol#252)
                - packages[5000] = Package(250,30) (contracts/NFTStakingContract.sol#253)
                - packages[8000] = Package(275,35) (contracts/NFTStakingContract.sol#254)
                - packages[50] = Package(200,20) (contracts/NFTStakingContract.sol#261)
                - packages[100] = Package(200,20) (contracts/NFTStakingContract.sol#262)
                - packages[500] = Package(225,25) (contracts/NFTStakingContract.sol#263)
                - packages[1500] = Package(225,25) (contracts/NFTStakingContract.sol#264)
                - packages[5000] = Package(250,30) (contracts/NFTStakingContract.sol#265)
                - packages[8000] = Package(250,30) (contracts/NFTStakingContract.sol#266)
        NFTStakingContract.packages (contracts/NFTStakingContract.sol#88) can be used in cross function reentranci
es:
```

```
        - NFTStakingContract.constructor(uint256,uint256,uint256) (contracts/NFTStakingContract.sol#142-159)
        - NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229)
        - NFTStakingContract.updateStage(uint256) (contracts/NFTStakingContract.sol#234-270)
        - users[user] = _user (contracts/NFTStakingContract.sol#224)
        NFTStakingContract.users (contracts/NFTStakingContract.sol#87) can be used in cross function reentrancies:
        - NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#293-330)
        - NFTStakingContract.initUser(address,uint256) (contracts/NFTStakingContract.sol#176-186)
        - NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229)
        - NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#195-206)
        - NFTStakingContract.userDetail(address) (contracts/NFTStakingContract.sol#336-338)
Reentrancy in NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#195-2
06):
        External calls:
        - lockReward(user,exchangeRate) (contracts/NFTStakingContract.sol#198)
                - success = nftRewardWallet.send(address(address(this)),rewardPTEK) (contracts/NFTStakingContract.
sol#217)
        - _tokenId = soulBoundNFT.safeMint(user,uri) (contracts/NFTStakingContract.sol#200)
        State variables written after the call(s):
        - users[user] = _user (contracts/NFTStakingContract.sol#204)
        NFTStakingContract.users (contracts/NFTStakingContract.sol#87) can be used in cross function reentrancies:
        - NFTStakingContract.dailyReward(address) (contracts/NFTStakingContract.sol#293-330)
        - NFTStakingContract.initUser(address,uint256) (contracts/NFTStakingContract.sol#176-186)
        - NFTStakingContract.lockReward(address,uint256) (contracts/NFTStakingContract.sol#212-229)
        - NFTStakingContract.mintNFT(address,uint256,string,bool,bool) (contracts/NFTStakingContract.sol#195-206)
        - NFTStakingContract.userDetail(address) (contracts/NFTStakingContract.sol#336-338)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

Functional Testing:

1) NFTCreationContract:-

A. Ensure that the contract allows the owner to successfully mint a unique NFT.

```
it("Should Allow Owner to Mint a New NFT Successfully", async () => {
        expect(await soulBoundNFT.balanceOf(addr1.address)).to.equal(0);
        await soulBoundNFT.connect(owner).safeMint(
              addr1.address,
              "external_uri_link"
            );
        expect(await soulBoundNFT.balanceOf(addr1.address)).to.equal(1);
        expect(await soulBoundNFT.tokenURI(0)).to.equal("external_uri_link");
      });
```

```
✔ Should Allow Owner to Mint a New NFT Successfully
```

B.Ensure that non-owners are prevented from minting a unique NFT.

```
it("Should Restrict Non-Owner from Minting NFT", async () => {
     await expect(
       soulBoundNFT.connect(addr1).safeMint(addr1.address, "external_uri_link")
     ).to.be.revertedWith(
       "error: this method can only be executed by the owner"
     );
   });
```

```
✔ Should Restrict Non-Owner from Minting NFT
```

Note: The current implementation of the minting process uses boolean inputs to determine if an NFT is classified as **Legendary** or **Rare**, instead of using an on-chain randomization mechanism.

PropTech comment- To support the PTEK business model and ensure metadata is added to NFTs before minting, the randomization process for rare and legendary attributes is performed on the backend. This approach ensures that the NFTs are properly assigned their attributes prior to being uploaded.

2) NFTStakingRewardWallet :-

A. It verifies that the NFTStakingRewardWallet smart contract can receive PTEK transactions and emit a BalanceReceived event as confirmation.

```
it("Should receive funds and emit BalanceReceived event", async () => {
  const amount = ethers.parseEther("1.0");
  expect(await ethers.provider.getBalance(await staking.nftRewardWallet())).to.equal(0);
    await expect(
    owner.sendTransaction({
      to: await staking.nftRewardWallet(),
      value: amount,
    })
  )
    .to.emit(nftRewardWallet, "BalanceReceived")
    .withArgs(owner.address, amount);
  expect(await ethers.provider.getBalance(await staking.nftRewardWallet())).to.equal(amount);
});
```

```
✔ Should receive funds and emit BalanceReceived event
```

B.The test verifies that the contract owner can successfully transfer PTEK from the contract's balance to a specified address (NFTStakingContract in this case), ensuring the recipient's balance increases by the correct amount.

```
it("Should send reward amount in PTEK to NFTStakingContract", async () => {
    const amount = ethers.parseEther("15.0");
    expect(
      await ethers.provider.getBalance(nftRewardWallet.target)
    ).to.equal(ethers.parseEther("10.0"));
    expect(await ethers.provider.getBalance(await staking.target)).to.equal(
      0
    );
    await expect(
      nftRewardWallet.connect(owner).send(staking.target,ethers.parseEther("1.0"))
    )
      .to.emit(nftRewardWallet, "NFTRewardSent")
      .withArgs(staking.target, ethers.parseEther("1.0"));
    expect(await ethers.provider.getBalance(await staking.target)).to.equal(
      ethers.parseEther("1.0")
    );
    expect(
      await ethers.provider.getBalance(nftRewardWallet.target)
    ).to.equal(ethers.parseEther("9.0"));
  });
```

```
✔ Should send reward amount in PTEK to NFTStakingContract
```

C. The test asserts that attempts by non-owners to invoke the send function are properly rejected.

```
it("Should not allow non owner to send amount in PTEK to NFTStakingContract", async () => {
    await expect(
      nftRewardWallet.connect(addr1).send(staking.target,ethers.parseEther("1.0"))
    ).to.be.revertedWith("error: this method can only be executed by the owner")
  });
```

```
✔ Should not allow non owner to send amount in PTEK to NFTStakingContract
```

D. The test asserts that sending rewards to zero addresses are properly rejected.

```
it("Should not allow sending amount to zero address", async () => {
    await expect(
      nftRewardWallet.connect(owner).send(ethers.ZeroAddress,ethers.parseEther("1.0"))
    ).to.be.revertedWith("error: invalid receiver address")
  });
```

```
✔ Should not allow sending amount to zero address
```

E.**Missing Function:** getTotalBalance(high)

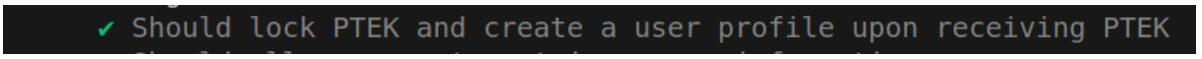
The NFTStakingRewardWallet contract lacks a function to retrieve the total PTEK balance of the NFT Staking Reward Wallet.

Status - Fixed by the developer.

3) NFTStakingContract:-

A. This test case checks the contract's functionality to lock PTEK and simultaneously create a user profile upon receiving PTEK

```
it("Should lock PTEK and create a user profile upon receiving PTEK", async () => {
  const amount = ethers.parseEther("1.0");
  expect(await ethers.provider.getBalance(staking.target)).to.equal(0);
  await expect(
    await addr1.sendTransaction({
      to: staking.target,
      value: amount,
    })
  )
    .to.emit(staking, "Received")
    .withArgs(addr1.address, amount);
  expect(await ethers.provider.getBalance(staking.target)).to.equal(amount);
  const userInfo = await staking.userDetail(addr1.address);
  expect(userInfo.packagePTEK).to.equal(amount);
  expect(userInfo.active).to.equal(true);
});
```



B. This test case checks that the NFTStakingContract allows its owner to retrieve user information, including current day, PTEK Locked and staking details.

```
it("Should allow owner to retrieve user information", async () => {
  const amount = ethers.parseEther("1.0");
  await addr1.sendTransaction({
    to: staking.target,
    value: amount,
  });
  const userInfo = await staking.connect(owner).userDetail(addr1.address);
  expect(userInfo.packagePTEK).to.equal(amount);
  expect(userInfo.active).to.equal(true);
});
```

```
✔ Should allow owner to retrieve user information
```

C.The test asserts that attempts by non-owners to retrieve user information are properly rejected.

```
it("Should fail if non-owner tries to retrieve user information", async () => {
   const amount = ethers.parseEther("1.0");
   await addr1.sendTransaction({
     to: staking.target,
     value: amount,
   });
   await expect(
     staking.connect(addr1).userDetail(addr1.address)
   ).to.be.revertedWith(
     "error: this method can only be executed by the owner"
   );
 });
```

```
✔ Should fail if non-owner tries to retrieve user information
```

D. This test case checks the contract's ability to receive the reward amount in PTEK from the NFT Staking Reward Wallet, based on the locked PTEK and the applicable reward percentage that is determined by the pack price and current stage.

Discrepancy:
As per the business rules, the reward should be triggered when the user sends PTEK to the NFT Staking Wallet from their wallet. However, the current implementation triggers the reward distribution when the owner mints the NFT for the user, which is not aligned with the specified requirements.

PropTech Comment-The NFT reward calculation for a user depends on the exchange rate of PTEK at the time of NFT purchase. As the user sends PTEK from his/her wallet to NFT Staking Contract (indicating purchase of NFT), it is not possible to receive exchange rate in that transaction. The next step, after receiving NFT PTEK from a user, is to mint NFT for that user, so the above mentioned problem has been solved in that step as the NFTmint function has all the necessary information for reward calculations along with minting NFT.

```
it("Should receive reward amount in PTEK from the NFT Staking Reward Wallet based on the locked PTEK and reward percentage",
   async () => {
    const amount = ethers.parseEther("15.0");
    expect(
      await ethers.provider.getBalance(await staking.nftRewardWallet())
    ).to.equal(0);
    await expect(
      owner.sendTransaction({
        to: await staking.nftRewardWallet(),
        value: amount,
      })
    )
      .to.emit(nftRewardWallet, "BalanceReceived")
      .withArgs(owner.address, amount);
    expect(
      await ethers.provider.getBalance(await staking.nftRewardWallet())
    ).to.equal(amount);
    await addr1.sendTransaction({
      to: staking.target,
      value: ethers.parseEther("50.0"),
    });
    await expect(
      staking.connect(owner).mintNFT(addr1.address, 10000, "uri", true, true)
    )
      .to.emit(nftRewardWallet, "NFTRewardSent")
      .withArgs(staking.target, ethers.parseEther("12.5"));
    expect(
      await ethers.provider.getBalance(await staking.nftRewardWallet())
    ).to.equal(ethers.parseEther("2.5"));
    expect(await ethers.provider.getBalance(await staking.target)).to.equal(
      ethers.parseEther("62.5")
    );
  });
```

```
    ✔ Should receive reward amount in PTEK from the NFT Staking Reward Wallet
based on the locked PTEK and reward percentage
```

E. This test case ensures that the mintNFT function enforces proper access control by reverting the transaction if a non-owner address attempts to mint an NFT. The test simulates a scenario where a non-owner (represented by addr1) tries to invoke the mintNFT function with specific parameters, and verifies that the transaction is reverted with the appropriate error message indicating that only the contract owner is authorized to perform this action.

```
it("Should fail if non-owner tries to mint Nft for user", async () => {
  await addr1.sendTransaction({
    to: staking.target,
    value: ethers.parseEther("50.0"),
  });
  await expect(
    staking.connect(addr1).mintNFT(addr1.address, 10000, "uri", true, true)
  ).to.be.revertedWith(
    "error: this method can only be executed by the owner"
  );
});
```

```
✔ Should fail if non-owner tries to mint Nft for user
```

F.To ensure that the staking smart contract correctly transitions between stages based on the total amount of PTEK locked and restricts the contract from accepting new members once the total locked amount reaches the defined stageMaxLimit. This test verifies both the correct progression through stages and the enforcement of limits on new member additions.

Note: The stage limits have been lowered for testing by setting values like **uint private stage2Limit = 20000 * weiDecimal;** instead of the original **2000000 * weiDecimal**. This modification enables faster and more efficient testing in the Hardhat environment, making it easier to simulate stage transitions without requiring large token amounts.

Steps:

1)Simulate the initial funding by sending Ether to the nftRewardWallet from the owner and additional addresses to prepare the contract to be able to transfer the reward amount to NFTStakingContract.

2)Perform transactions(transfer PTEK from user wallet to NFTStakingContract) and mint NFTs for different addresses.
Verify that the contract transitions through stages 1, 2, 3, and 4 correctly based on the total PTEK locked.

3)Attempt to add new members after reaching the maximum limit and ensure the contract rejects further additions with the appropriate error message.

```
 it("Validate Stage Transition Based on Total PTEK Locked and
restrict contract to accepting new members once the total PTEK
locked reaches stageMaxLimit", async () => {
   await owner.sendTransaction({
     to: await staking.nftRewardWallet(),
     value: ethers.parseEther("9500.0"),
   });
   await addrs[11].sendTransaction({
     to: await staking.nftRewardWallet(),
     value: ethers.parseEther("9500.0"),
   });
   await addrs[12].sendTransaction({
     to: await staking.nftRewardWallet(),
     value: ethers.parseEther("9500.0"),
   });
   await addrs[13].sendTransaction({
     to: await staking.nftRewardWallet(),
     value: ethers.parseEther("9500.0"),
   });
   expect(await staking.currentStage()).to.equal(1);
   await addr1.sendTransaction({
     to: staking.target,
     value: ethers.parseEther("8000.0"),
   });
   await staking
     .connect(owner)
     .mintNFT(addr1.address, 10000, "uri", true, true);
   expect(await staking.currentStage()).to.equal(1);
   await addr2.sendTransaction({
```

```
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addr2.address, 10000, "uri", true, true);
  expect(await staking.currentStage()).to.equal(2);
  await addrs[0].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await addrs[1].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[0].address, 10000, "uri", true, true);
  await staking
    .connect(owner)
    .mintNFT(addrs[1].address, 10000, "uri", true, true);
  await addrs[2].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  expect(await staking.currentStage()).to.equal(2);
  await staking
    .connect(owner)
    .mintNFT(addrs[2].address, 10000, "uri", true, true);
  expect(await staking.currentStage()).to.equal(3);
  await addrs[3].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await addrs[4].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[3].address, 10000, "uri", true, true);
  await staking
    .connect(owner)
```

```
    .mintNFT(addrs[4].address, 10000, "uri", true, true);
  await addrs[5].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[5].address, 10000, "uri", true, true);
  expect(await staking.currentStage()).to.equal(3);
  await addrs[6].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[6].address, 10000, "uri", true, true);
  expect(await staking.currentStage()).to.equal(4);
  await addrs[7].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await addrs[8].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[7].address, 10000, "uri", true, true);
  await staking
    .connect(owner)
    .mintNFT(addrs[8].address, 10000, "uri", true, true);
  expect(await staking.currentStage()).to.equal(4);
  await addrs[9].sendTransaction({
    to: staking.target,
    value: ethers.parseEther("8000.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addrs[9].address, 10000, "uri", true, true);
  await expect(
    addrs[10].sendTransaction({
      to: staking.target,
      value: ethers.parseEther("8000.0"),
```

```
    })
  ).to.be.revertedWith("error: new members limit reached");
});
```

```
    ✔ Validate Stage Transition Based on Total PTEK Locked and restrict contract
to accepting new members once the total PTEK locked reaches stageMaxLimit (92ms)
```

G.This test case evaluates the NFTStakingContract functionality to prevent the owner from updating the PTEK rate and total global PTEK locked within 24 hours of the previous update. It ensures that updates to these parameters can only be performed after a 24-hour waiting period. Initially, the simulation has the owner update the PTEK rate and global locked PTEK via the dailyLocking function, recording the dailyLockTime for the next update. The test then attempts another update within 24 hours, which should revert with an error message. After simulating 24 hours passing with moveBlocks, the owner successfully updates the PTEK rate, validating the 24-hour restriction mechanism.

```
it("Should not allow owner to update PTEK rate and global ptek locked within 24 hour of the previous update",
 async () => {
  let oldLockTime = await staking.dailyLockTime();
  // await staking.connect(owner).dailyReward(addr1.address);
  await staking.connect(owner).dailyLocking(10000, 100000);
  expect((await staking.dailyLockTime()) - oldLockTime).to.equal(
    24 * 60 * 60
  );
  await expect(
    staking.connect(owner).dailyLocking(10000, 100000)
  ).to.be.revertedWith("error: wait atleast 24 hours for next update!");
  await moveBlocks(24 * 60 * 60);
  await staking.connect(owner).dailyLocking(10000, 100000);
});
```

```
✔ Should not allow owner to update PTEK rate and global ptek locked within 24 hour of the previous update
```

H.**[FAILED]**This test simulates the reward distribution mechanism, where a user should receive PTEK daily reward based on their locked PTEK and Daily Unlocking Allowance percentage from the NFT Staking Reward Wallet. However, the contract fails due to the below condition in dailyReward function:**(**high**)**

```
require(users[user].lastPaid + 1 days >= dailyLockTime, "error: daily reward already unlocked for today");
```

The current timestamp is used in the dailyLockTime constructor variable during deployment, but this approach is not functioning as expected. Also tried with a timestamp of 12:01 AM for the same day while deploying the contract,but it too failed.

The above condition always evaluates to false, preventing the dailyReward function from executing as expected, even though the dailyLocking function successfully completes.

```
it("Should receive reward amount in PTEK from the NFT Staking Reward Wallet based on the locked PTEK and reward percentage",
 async () => {
  const amount = ethers.parseEther("15.0");
  await owner.sendTransaction({
    to: await staking.nftRewardWallet(),
    value: amount,
  });
  await addr1.sendTransaction({
    to: staking.target,
    value: ethers.parseEther("50.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(addr1.address, 10000, "uri", true, true);
  let oldBalance = await ethers.provider.getBalance(addr1.address);
  console.log(oldBalance);
  await staking.connect(owner).dailyReward(addr1.address);
  await staking.connect(owner).dailyLocking(10000, 100000);
  await staking.connect(owner).dailyReward(addr1.address);
  let newBalance = await ethers.provider.getBalance(addr1.address);
  console.log(newBalance);
});
```

```
    Should receive reward amount in PTEK from the NFT Staking Reward Wallet based on the locked PTEK and reward percentage:
  Error: VM Exception while processing transaction: reverted with reason string 'error: daily reward already unlocked for today'
at NFTStakingContract.dailyReward (contracts/NFTStakingContract.sol:202)
at EdrProviderWrapper.request (node_modules/hardhat/src/internal/hardhat-network/provider/provider.ts:433:41)
at HardhatEthersSigner.sendTransaction (node_modules/@nomicfoundation/hardhat-ethers/src/signers.ts:125:18)
at send (node_modules/ethers/src.ts/contract/contract.ts:313:20)
at Proxy.dailyReward (node_modules/ethers/src.ts/contract/contract.ts:352:16)
at Context.<anonymous> (test/NftStaking.js:256:5)
```

Another Scenario:

If dailyLockTime is set up to 1 day (i.e., not greater than 86400 seconds)

When dailyLockTime is configured for up to 1 day (not exceeding 86400 seconds) in the constructor during deployment, the dailyReward executes successfully, but a critical issue arises. The dailyReward function can be invoked multiple times within a single day without updating dailyLockTime, which contradicts the intended contract behavior. Even though only the owner can call this function, it should still restrict the owner to one call per day.

Moreover, the owner can continuously update dailyLockTime without waiting for 24 hours between updates. This bypasses the intended 24-hour restriction, allowing the owner to make multiple updates within the same day. Both of these behaviors are unexpected and need to be addressed to ensure proper time-bound reward distributions and updates.

Status - Recommendation is to have the suggested flow for improvisation.

I.**Incorrect Daily Unlocking Allowance Logic(**high**)**

The contract logic for calculating the Daily Unlocking Allowance Percentage is flawed.
According to business rules,Daily Unlocking Allowance % should be calculated as:

Daily Unlocking Allowance % =
IF(PTEK Price Current <= PTEK Price Start, Max Daily Unlocking Allowance,
(PTEK Price Start * Max Daily Unlocking Allowance/PTEK Price Current))

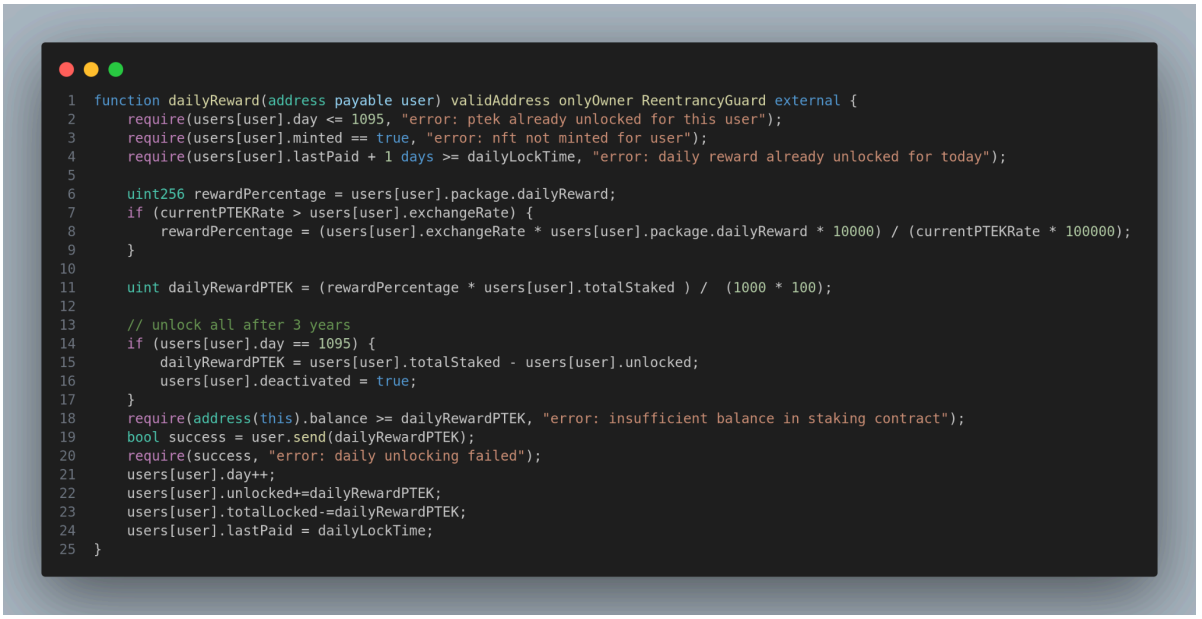
However, the contract instead uses the following logic

```
uint256 rewardPercentage = users[user].package.dailyReward;
        if (currentPTEKRate > users[user].exchangeRate) {
            rewardPercentage = (users[user].exchangeRate * users[user].package.dailyReward * 10000) / (currentPTEKRate * 100000);
        }

```

Status - PropTech has reviewed and verified the formula for calculating the Daily Unlocking Allowance Percentage, confirming its accuracy. Testing of the daily distribution process has been successfully conducted, with the process functioning as intended and no issues encountered. PropTech affirms that the implemented logic aligns with the business requirements and performs effectively in practice.

J.Missing Logic in dailyReward function:(high)

The contract lacks a mechanism to deactivate users who have unlocked their total locked amount (staked + reward) within the specified 3-year period. As a result, users can continue unlocking rewards until the full 3-year period, even if they have already surpassed their total staked amount, as long as the contract maintains a sufficient balance. This oversight allows for potential over-unlocking and must be addressed to ensure users cannot withdraw more than their entitled amount within the defined time frame.



```
function dailyReward(address payable user) validAddress onlyOwner ReentrancyGuard external {
    require(users[user].day <= 1095, "error: ptek already unlocked for this user");
    require(users[user].minted == true, "error: nft not minted for user");
    require(users[user].lastPaid + 1 days >= dailyLockTime, "error: daily reward already unlocked for today");

    uint256 rewardPercentage = users[user].package.dailyReward;
    if (currentPTEKRate > users[user].exchangeRate) {
        rewardPercentage = (users[user].exchangeRate * users[user].package.dailyReward * 10000) / (currentPTEKRate * 100000);
    }

    uint dailyRewardPTEK = (rewardPercentage * users[user].totalStaked ) /  (1000 * 100);

    // unlock all after 3 years
    if (users[user].day == 1095) {
        dailyRewardPTEK = users[user].totalStaked - users[user].unlocked;
        users[user].deactivated = true;
    }
    require(address(this).balance >= dailyRewardPTEK, "error: insufficient balance in staking contract");
    bool success = user.send(dailyRewardPTEK);
    require(success, "error: daily unlocking failed");
    users[user].day++;
    users[user].unlocked+=dailyRewardPTEK;
    users[user].totalLocked-=dailyRewardPTEK;
    users[user].lastPaid = dailyLockTime;
}
```

Suggested Fixes

- Introduce logic to deactivate users once their total locked amount has been unlocked within the 3-year period.

Status - Fixed by the developer.

K. **Missing Function:** getTotalBalance(high)

The NFTStakingContract contract lacks a function to retrieve the total PTEK balance of the NFTStakingContract.

Status - Fixed by the developer.

L. This test case verifies that the owner can call payPendingReward for the interval between the contract's creation time and the current deployment time. Initially, the owner funds the nftRewardWallet and stakes an amount using the staking contract. The owner then mints an NFT for addr1 with a timestamp that is 5 days in the past.

The test checks the balance of addr1 before making multiple calls to payPendingReward, ensuring rewards are accumulated correctly. After six successful calls, it attempts an additional call, which should revert with an error message indicating that pending rewards have already been paid.

Next, the test verifies that the balance of addr1 reflects the correct amount earned. The owner then invokes dailyLocking, and the test expects an event indicating that daily rewards have been unlocked. An attempt to call dailyReward again on the same day should revert with an appropriate error message.

Finally, after simulating the passage of 24 hours using moveBlocks, the owner calls dailyLocking again and successfully unlocks daily rewards, confirming that the time restriction is enforced properly.

```
it("It should allow owner to call payPendingReward for the interval between contract's creation time and current time(deployment)",
    async () => {
  await owner.sendTransaction({
    to: await staking.nftRewardWallet(),
    value: ethers.parseEther("15.0"),
  });
  await addr1.sendTransaction({
    to: staking.target,
    value: ethers.parseEther("50.0"),
  });
  await staking
    .connect(owner)
    .mintNFT(
      addr1.address,
      (await now()) - 432000,
      50,
      10000,
      "uri",
      true,
      true
    );
  let oldBalance = await ethers.provider.getBalance(addr1.address);
  for (let i = 0; i < 6; i++) {
    await staking.connect(owner).payPendingReward(addr1.address, 10000);
  }
  await expect(
    staking.connect(owner).payPendingReward(addr1.address, 10000)
  ).to.be.revertedWith("error: pending rewards already paid");
  let newBalance = await ethers.provider.getBalance(addr1.address);
  expect(newBalance - oldBalance).to.equal(ethers.parseEther("0.84375"));
  await staking.connect(owner).dailyLocking(10000, 100000);
  await expect(staking.connect(owner).dailyReward(addr1.address)).to.emit(
    staking,
    "DailyRewardUnlocked"
  );
  await expect(
    staking.connect(owner).dailyReward(addr1.address)
  ).to.be.revertedWith("error: daily reward already unlocked for today");
  await moveBlocks(86400);
  await staking.connect(owner).dailyLocking(10000, 100000);
  await expect(staking.connect(owner).dailyReward(addr1.address)).to.emit(
    staking,
    "DailyRewardUnlocked"
  );
});
```

```
    ✔ It should allow owner to call payPendingReward for the interval
 between contract's creation time and current time(deployment) (12140
ms)
```

Proposed Improvements to the payPendingReward Function Logic-

The payPendingReward function requires enhancements to effectively manage the final day of a user's reward period (day 1095). Specifically, the function should implement logic to unlock the remaining rewards by setting dailyRewardPTEK to _user.totalStaked - _user.unlocked and deactivate the user after they have claimed their rewards. Additionally, to prevent unnecessary reverts during the last iteration, the function should check if dailyRewardPTEK exceeds _user.totalLocked. If it does, the function should adjust dailyRewardPTEK to match _user.totalLocked and deactivate the user.

While this scenario is unlikely to occur unless there is a significant delay (e.g., more than 444 days) in deploying the contracts, including this logic is important for ensuring robustness and preventing potential issues in reward distribution.

NOTE-As discussed in the meeting, user onboarding may commence prior to the contract going live. In this scenario, the pending reward function will be utilized to distribute rewards for the interval between the contract's creation time (during onboarding) and the contract deployment time. However, users will be required to make a deposit once the contract is live.

Appendix

Finding's Categories:

Reentrancy Risk

Reentrancy Risk findings identify vulnerabilities where external calls could lead to unwanted re-entry into contract functions, potentially compromising contract integrity and leading to unintended behavior or asset loss.

Gas Optimization

Gas Optimization findings highlight areas where the contract's operations could be made more gas-efficient, either through optimizing storage access patterns or restructuring code to minimize computational overhead, thereby reducing transaction costs.

Arithmetic Operations

Arithmetic Precision Loss findings pinpoint instances where the order of mathematical operations could lead to significant precision loss, affecting the contract's financial calculations and potentially leading to incorrect outcomes.

Event Logging Practices

Event Logging Practices findings review the contract's use of events for transparency and tracking, identifying opportunities to enhance visibility into contract operations and critical actions for off-chain monitoring and interfacing.

Compiler Versioning

Compiler Versioning findings emphasize the importance of specifying a fixed compiler version to avoid unpredictable behavior or vulnerabilities introduced by compiler updates, ensuring consistent and secure contract compilation.

Solidity Best Practices

Solidity Best Practices findings include recommendations for adhering to established coding conventions and practices unique to Solidity development, aiming to improve code security, readability, and maintainability.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) outlined in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services outlined in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions outlined in the Agreement. This report may not be transmitted, disclosed, referred to, or relied upon by any person for any purposes without **SoluLab**'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 considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts **SoluLab** to perform a security assessment. This report 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 in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessment process intended 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. **SoluLab**'s position is that each company and individual is responsible for their due diligence and continuous security.

SoluLab'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