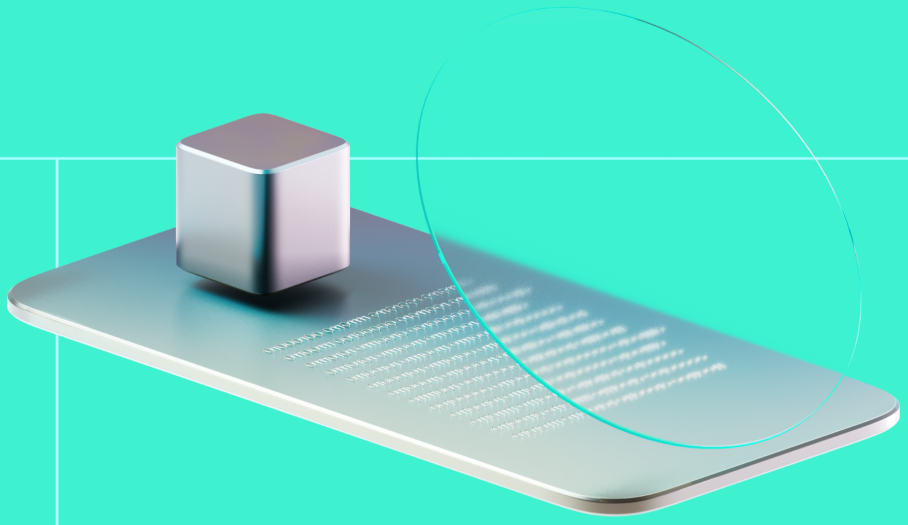
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Frens

Date: 12 Dec, 2023

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

We thank Frens for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

Frens is an innovative staking platform designed to democratize Ethereum staking, making it accessible, social, and enjoyable for groups of friends through user-friendly decentralized pools and NFT-based staking participation.

Platform: EVM

Language: Solidity

Tags: Staking Pools, ERC721

Timeline: 20.11.2023 - 12.12.2023

Methodology: Link

Last review scope

Repository	https://github.com/frens-pool/frens-contracts-v2
Commit	413d30f

View full scope

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Audit Summary

10/10	9/10	71.62%	8/10
Security score	Code quality score	Test coverage	Documentation quality score

Total: 8.5/10

The system users should acknowledge all the risks summed up in the risks section of the report.

3	3	0	0
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	0	0	0	0
High	0	0	0	0
Medium	0	2	0	0
Low	0	1	0	0

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Document

Name	Smart Contract Code Review and Security Analysis Report for Frens
Approved By	Grzegorz Trawiński | SC Audits Expert at Hacken OÜ
Audited By	Ivan Bondar | SC Lead Auditor at Hacken OÜ
Website	https://www.frens.fun/
Changelog	24.11.2023 – Preliminary Report 12.12.2023 – Final Report

Introduction

Hacken OÜ (Consultant) was contracted by Frens (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

The Frens Staking Protocol is an innovative Ethereum staking solution that allows users to participate in Ethereum staking pools with smaller ETH amounts. By leveraging the pooling mechanism, users who may not possess the full 32 ETH required for individual staking can still participate and earn rewards. The protocol issues NFTs representing each participant's share in the staking pool, offering a unique blend of staking and NFT functionalities.

Contracts in Audit Scope:

- StakingPool.sol — Central to the protocol, this contract enables users to join staking pools. It facilitates the collection of ETH, stakes 32 ETH to the Ethereum 2.0 Deposit Contract, and manages the rewards and withdrawals. Features:
    - Allows deposits to the pool with ETH and issues corresponding NFTs representing each user's share.
    - Manages the staking process, including submitting the validator's public key, signature, and credentials.
    - Supports withdrawal of funds and claiming of rewards by NFT holders.
- FrensPoolShare.sol - An ERC-721 (NFT) contract representing individual shares in a staking pool. Features:
    - Each NFT corresponds to a user's share in a specific staking pool.

    - Supports standard ERC-721 functionalities like transfers, with additional checks for locked shares.
    - Integrates with the StakingPool contract to reflect the staking positions and rewards.
- IFrensPoolShareTokenURI.sol — Interface for generating and managing token URIs for the NFTs issued.
- IFrensArt.sol — Interface for integrating art or metadata related to the NFTs issued by the protocol.
- IFrensOracle.sol, IFrensStorage.sol — Interfaces for additional protocol functionalities, including storage and oracle services.

Privileged roles

- StakingPool:
    - Owner:
        - Ability to stake 32 ETH to the deposit contract, including setting validator information.
        - Authority to change the art for the NFTs in the pool.
- FrensPoolShare:
    - Staking Pool:
        - Has the authority to mint new NFTs representing staking shares.
        - Possesses the power to burn these NFTs, controlling their lifecycle.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **8** out of **10**.

- Functional requirements:
  - Project overview is detailed.
  - Use cases are described and detailed.
  - All interactions are described.
  - Documentation-Implementation Mismatch Notice (I09)
- Technical description is robust:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.

Test coverage

Code coverage of the project is **71.62%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Not all branches are covered by test cases.

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Security score

As a result of the audit, the code contains **no** issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **8.5**. The system users should acknowledge all the risks summed up in the risks section of the report.

Risks

Risk Statement

This audit report focuses exclusively on the security assessment of the contracts within the specified review scope. Interactions with out-of-scope contracts are presumed to be correct and are not examined in this audit. We want to highlight that Interactions with contracts outside the specified scope, such as:

1. ./contracts/FrensArt.sol
2. ./contracts/FrensLogo.sol
3. ./contracts/FrensMetaHelper.sol
4. ./contracts/FrensOracle.sol
5. ./contracts/FrensPoolShareTokenURI.sol
6. ./contracts/FrensStorage.sol
7. ./contracts/PmFont.sol
8. ./contracts/Waves.sol
9. ./contracts/interfaces/IENS.sol
10. ./contracts/interfaces/IFrensLogo.sol
11. ./contracts/interfaces/IFrensMetaHelper.sol
12. ./contracts/interfaces/IPmFont.sol
13. ./contracts/interfaces/IReverseResolver.sol
14. /contracts/interfaces/IWaves.sol

have not been verified or assessed as part of this report.

While we have diligently identified and mitigated potential security risks within the defined scope, it is important to note that our assessment is confined to the isolated contracts within this scope. The overall security of the entire system,

including external contracts and integrations beyond our audit scope, cannot be guaranteed.

Users and stakeholders are urged to exercise caution when assessing the security of the broader ecosystem and interactions with external contracts. For a comprehensive evaluation of the entire system, additional audits and assessments outside the scope of this report are necessary.

This report serves as a snapshot of the security status of the audited contracts within the specified scope at the time of the audit. We strongly recommend ongoing security evaluations and continuous monitoring to maintain and enhance the overall system's security.

Other risks:

- Importance of Pool Owner's Authority:
    - Setting Validator Information:
        - The pool owner is responsible for configuring essential validator parameters, including the public key, withdrawal credentials, signature, and deposit data root. These settings are pivotal for the validator’s role in Ethereum's PoS staking.
    - Validation Locked State:
        - Once the validator information is set and if the validatorLocked condition is active, these parameters become immutable. This inflexibility highlights the criticality of the pool owner’s initial setup.
    - Trust and Due Diligence:
        - **Participants must place significant trust in the pool owner’s expertise and integrity**. Mismanagement or malfeasance in setting these parameters can have irreversible consequences.

- Unfinalized RageQuit and Absent Burn Functionality in Staking Pool Contract:
    - The current implementation of the staking pool contract includes a feature named "RageQuit", which is not active in the existing version of the contract. Consequently, any mechanisms or controls expected from the RageQuit feature are not in effect. Furthermore, the associated FrensPoolShare NFT contract includes a burn function, allowing the burning of NFTs by the pool that minted them. However, this burn functionality is not reflected in the current StakingPool contract.
    - New versions of the protocol may activate the RageQuit functionality or introduce a burn mechanism in the StakingPool contract, potentially leading to significant changes in the pool's operation and user participation. **Users are advised to thoroughly research and verify the version of the pool contract** they interact with, including understanding how the RageQuit functionality and NFT burning might be implemented or activated in future versions.
    - Before engaging in staking activities within the pool, it is crucial for users to thoroughly review and understand key pool parameters. This is especially pertinent concerning the fee recipient address, as the transfer of fees is an integral part of the claim function in the contract.

Findings

Critical

No critical severity issues were found.

High

No high severity issues were found.

Medium

M01. Insecure Setting of DepositContractAddress in Staking Function

Impact	High
Likelihood	Low

In the StakingPool contract, the *_stake* function dynamically retrieves the *depositContractAddress* from the *frensStorage* at the time of staking. This approach poses potential security risks due to the lack of a fixed, verified address for deposit transactions.

The function *_stake* is responsible for facilitating the staking of Ether in the contract. It is crucial that the address to which the funds are sent (the *depositContractAddress*) is secure and verified.

Currently, the contract fetches this address from *frensStorage* during staking operation. This design means that any changes to *frensStorage* can alter the

depositContractAddress, potentially leading to unforeseen security vulnerabilities, including misdirection of funds.

Affected Code:

```
function _stake() internal {
    // ... other checks ...

    address depositContractAddress =
frensStorage.getAddress(keccak256(abi.encodePacked("external.contract.address",
"DepositContract")));
    currentState = PoolState.staked;
    IDepositContract(depositContractAddress).deposit{value: 32 ether}(
        pubKey,
        withdrawal_credentials,
        signature,
        deposit_data_root
    );
    // ... rest of the function ...
}
```

The dynamic setting of the *depositContractAddress* increases the risk of funds being sent to an incorrect or malicious address. This not only jeopardizes the security of the funds being staked but also undermines the trust in the staking process. In the worst case, this could lead to the loss of all funds sent to the contract.

Path: ./contracts/StakingPool.sol : _stake()

Recommendation: It is advisable to set the *depositContractAddress* during the deployment of the StakingPool contract and ensure that it is immutable. This can be achieved by either hardcoding the address in the contract or setting it through the constructor and not allowing further modifications.

Found in: 6b73ad4

Status: Fixed (Revised commit: 27e7aa6)

Remediation: The revised implementation of the *_stake* function now utilizes a fixed *depositContract* address set during the contract's construction. This approach eliminates the risk associated with dynamically fetching the address and ensures the stability and security of the deposit transactions.

M02. Possible Manipulation and Limit Exceedance of Fee Percentage in Reward Calculations

Impact	High
Likelihood	Low

The *claim* and *getDistributableShare* functions in the StakingPool contract retrieve the fee percentage dynamically from storage (*frensStorage*). This approach introduces risks related to the manipulation of fee percentages and the potential for these percentages to exceed logical limits.

In both functions, the fee percentage is fetched using `frensStorage.getUint(keccak256(abi.encodePacked("protocol.fee.percent")));`

Since this value is retrieved from storage each time, it can be subject to changes that might occur between the initiation of a transaction and its execution (front running).

Additionally, there is no check in place to ensure that the fee percentage does not exceed 100%, which could lead to illogical and erroneous fee calculations.

The lack of fixed fee percentages and the absence of upper limit checks expose users to risks of unfair fee deductions or even total depletion of their rewards

due to excessively high fees. This issue can undermine the integrity of the reward distribution mechanism.

Path: ./contracts/StakingPool.sol : claim(), getDistributableShare()

Recommendation: To address the issue of dynamic fee percentage manipulation and the risk of exceeding logical limits, the following steps are recommended:

- Set Fee Percentage During Deployment:
    - Establish the fee percentage as a fixed value during the deployment of the StakingPool contract. This should be done through the constructor or a similar initialization function, ensuring the fee percentage remains immutable throughout the contract's lifecycle.
- Implement Reasonable Upper Limit for Fee Percentage:
    - Introduce a validation check within the contract to ensure the fee percentage does not surpass a reasonable upper limit, which should be significantly lower than 100%. This limit should be determined based on the contract's economics and fairness considerations, ensuring that the fee structure remains logical and equitable for all participants.

Found in: 6b73ad4

Status: Fixed (Revised commit: 1da1be7)

Remediation: The new implementation addresses these concerns by setting the fee percentage (*feePercent*) during the contract's construction. This static approach ensures that the fee percentage remains constant and predictable

throughout the contract's lifecycle. Furthermore, the fee percentage is capped at a maximum of 10%, thereby eliminating the risk of excessive fees.

Low

L01. Potential DoS Risk in claim() Due to Fee Recipient Handling

Impact	Medium
Likelihood	Low

The *claim* function in the StakingPool contract includes a mechanism to transfer fees to a designated recipient. This process involves an external call to transfer Ether to the fee recipient's address. If this transfer fails (e.g., due to the recipient's contract throwing an error), it can block the entire *claim* function, leading to a Denial of Service (DoS) situation.

The issue within the *claim* function of the StakingPool contract centers around the way the *feeRecipient* is determined and how Ether is transferred to this address. The critical code sections are as follows:

- Acquiring the Fee Recipient Address:

```
address feeRecipient =
frensStorage.getAddress(keccak256(abi.encodePacked("protocol.fee.recipient")));
```

This line retrieves the fee recipient's address from the frensStorage contract. The process relies on the assumption that the address stored in frensStorage is always valid and capable of receiving Ether.

- Transferring Ether to Fee Recipient:

```
if (feePercent > 0 && !exited) {
    uint feeAmount = (feePercent * amount) / 100;
    if (feeAmount > 1){
        (bool success1, /*return data*/) = feeRecipient.call{value: feeAmount -
1}(""); //-1 wei to avoid rounding error issues
        assert(success1);
    }
    amount = amount - feeAmount;
}
```

The contract calculates the fee amount based on the reward and then attempts to transfer it to the feeRecipient address using a low-level call. If this call fails, the entire claim function reverts.

This vulnerability poses a risk to the functionality of the claim process, as users would be unable to access their rewards if the fee transfer fails. It compromises the reliability of the staking pool.

Path: ./contracts/StakingPool.sol : claim()

Recommendation: To mitigate the risk of Denial of Service (DoS) in the claim function due to the current fee transfer mechanism, two solutions are proposed:

- Establish Fee Recipient at Deployment:
  - Configure the contract to set the *feeRecipient* address during its deployment. This ensures that the address is fixed and known to users beforehand, allowing them to verify its validity before interacting with the contract.
  - By fixing the *feeRecipient* at deployment, users gain transparency about where fees are directed, enhancing trust in the contract.
- Decouple Fee Processing from Reward Claims:
  - Modify the contract to track owed fees separately within the contract's state. Instead of immediately transferring the fee during

the claim process, record the fee amount owed to the fee recipient. This could be implemented using a mapping to track fees owed to each recipient address.

- Implement a new function that allows the fee recipient to independently withdraw their accumulated fees. This function should safely handle the transfer of the owed fees, ensuring that failures in fee collection do not impact the users' ability to claim rewards.
- Given the proposed change in the fee handling mechanism, the reward calculation logic in `_getShare` needs to be updated to accurately reflect the net rewards due to each user, excluding the fees. Ensure that the reward calculation takes into account the separation of reward and fee accounting to maintain accurate tracking of distributable rewards.

Found in: 6b73ad4

Status: Fixed (Revised commit: 1da1be7)

Remediation: The revised implementation of the `claim` function addresses this vulnerability by establishing the fee recipient address during contract deployment, and not dynamically fetching it during each operation. Users can verify this address of the fee recipient before participating in the staking pool. This change enhances the security and predictability of the fee transfer process.

Informational

I01. Floating Pragma

The contract employs a floating pragma statement, *pragma solidity >=0.8.0 <0.9.0;*, which allows for compilation with any version between 0.8.0 and 0.9.0.

The specified pragma directive does not lock the contract to a specific compiler version. Instead, it permits compilation with any minor release within the given range. Different minor versions of the Solidity compiler may introduce changes or optimizations that affect contract execution or introduce unforeseen vulnerabilities.

The primary risk lies in the potential for unintended behavior or vulnerabilities due to changes in compiler versions. This can lead to issues with contract functionality or security, particularly if a newer compiler version includes changes not accounted for in the contract's design.

Paths: ./contracts/

Recommendation: Adopt a fixed version pragma that aligns with the specific compiler version used during the contract's testing and development phase. This fixed version pragma ensures that the contract is always compiled with a known, stable compiler version, reducing the risk of unexpected behavior or vulnerabilities introduced by compiler updates.

Found in: 6b73ad4

Status: Fixed (Revised commit: fd81615)

Remediation: The contract now employs a fixed pragma statement *pragma solidity 0.8.20;*, replacing the previous floating pragma.

I02. Missing Explicit Visibility Declarations in StakingPool.sol and FrensPoolShare.sol Contract State Variables

In the StakingPool.sol and FrensPoolShare.sol contracts, certain state variables lack explicit visibility declarations. Specifically, *currentState* in StakingPool.sol and *frensStorage* in FrensPoolShare.sol are missing these declarations.

Solidity defaults to internal visibility when it is not explicitly stated. While this may be appropriate in many cases, relying on default behavior can lead to misunderstandings or maintenance challenges.

The lack of explicit visibility declarations may not immediately lead to security risks but poses concerns regarding code clarity and maintainability. It can also lead to ambiguity in how these variables are intended to be accessed, potentially causing issues in future updates or modifications.

Paths: ./contracts/FrensPoolShare.sol

./contracts/StakingPool.sol

Recommendation: Update the *currentState* variable in StakingPool.sol and the *frensStorage* variable in FrensPoolShare.sol with explicit visibility declarations.

Found in: 6b73ad4

Status: Fixed (Revised commit: 1da1be7)

Remediation: State variables *currentState* in StakingPool.sol and *frensStorage* in FrensPoolShare.sol now have explicit *public* visibility declarations.

I03. Constructors of FrensPoolShare and StakingPool Contracts Lack Null Address Validation

In the FrensPoolShare and StakingPool contracts, the constructors do not include checks to validate that input addresses are non-zero.

- In FrensPoolShare, the constructor accepts *IFrensStorage frensStorage_* as an input and directly assigns it to the state variable without checking if it is the zero address.
- In StakingPool, similar behavior is observed. The constructor takes *address owner_*, and *IFrensStorage frensStorage_* as inputs. While *owner_* and *frensStorage_* are assigned to state variables, no zero address validation is performed.
- Additionally, in StakingPool, derived addresses for *artForPool* and *frensPoolShare* are obtained from frensStorage but are not validated against being zero.

Assigning a zero address to critical components in these contracts could render certain functionalities inoperative, potentially leading to operational failures or security weaknesses.

Paths: ./contracts/FrensPoolShare.sol : constructor()

./contracts/StakingPool.sol : constructor()

Recommendation: Implement events in all functions where state changes occur or significant actions are performed.

Found in: 6b73ad4

Status: Fixed (Revised commit: 1da1be7)

Remediation: The new implementation introduces zero address checks for input addresses in the constructors of FrensPoolShare and StakingPool.

I04. Absence of Events in Key Functions of StakingPool Contract

The StakingPool contract's key functions – *addToDeposit*, *_setPubKey*, *withdraw*, and *claim* – do not emit events.

Events are crucial in Ethereum contracts, especially for functions that alter the state, as they facilitate tracking and logging of state changes on the blockchain. This is particularly important for functions that handle staking-related activities, as they are integral to user interactions and financial transactions within the protocol.

Without events, it becomes challenging to track when and how these functions are called, hindering transparency.

Path: ./contracts/StakingPool.sol : addToDeposit(), _setPubKey(), withdraw(), claim()

Recommendation: Implement event emissions in the aforementioned functions. The events should be designed to log all pertinent information related to the actions taken. For instance, in *addToDeposit*, the event should log the NFT ID and the amount of funds added. In *setPubKey*, the event should capture the new validator information being set. For the *withdraw* and *claim* functions, events should include details like the NFT ID, the amount involved, and the address of the participant executing the action.

Found in: 6b73ad4

Status: Fixed (Revised commit: c277159)

Remediation: Key functions within the StakingPool contract, such as *addToDeposit*, *_setPubKey*, *withdraw*, and *claim*, now emit events.

I05. Unconventional Encoding Method in _toWithdrawalCred() Function

The function *_toWithdrawalCred* aims to generate withdrawal credentials, which are crucial for the secure withdrawal of funds in Ethereum's staking protocol.

The current implementation converts the address *a* to uint256 (*uintFromAddress*) and adds a large constant to it. This method, although achieving the intended format, is not a standard approach and can lead to confusion.

```
function _toWithdrawalCred(address a) private pure returns (bytes memory) {
    uint uintFromAddress = uint256(uint160(a));
    bytes memory withdralDesired = abi.encodePacked(
        uintFromAddress +
         0x0100000000000000000000000000000000000000000000000000000000000000
    );
    return withdralDesired;
}
```

While the current implementation may technically achieve the desired outcome, its unconventional nature can lead to misunderstandings and maintenance challenges.

Path: ./contracts/StakingPool.sol : _toWithdrawalCred()

Recommendation: It is recommended to modify the *_toWithdrawalCred* function to use a more standard encoding method. Specifically, use `abi.encodePacked(bytes1(0x01), bytes11(0x0), address(a));`

This change will enhance the clarity and maintainability of the code.

Found in: 6b73ad4

Status: Fixed (Revised commit: 76f513e)

Remediation: The updated implementation adheres to a more standard and clear method of encoding withdrawal credentials.

I06. Unused Ownable Functionality in Smart Contract

The FrensPoolShare contract imports Ownable from OpenZeppelin's contracts but does not appear to utilize its functionality. The Ownable contract provides a basic access control mechanism, with an account (the owner) that can be granted exclusive access to specific functions. Including this import without using its features can lead to unnecessary code bloat and potential confusion about the contract's access control structure.

While not directly a security risk, including unused imports can contribute to the complexity and potential maintenance burden.

Paths: ./contracts/FrensPoolShare.sol

Recommendation: Review the contract to confirm whether the Ownable functionality is needed. If it is not being utilized, remove the import to streamline the contract.

Found in: 6b73ad4

Status: Mitigated (The poolShare NFT contract needs to be ownable to allow us to edit the OpenSea page for the NFT, even though this is not used in the contract itself. Nothing modified for this issue.)

Remediation: Based on the client's requirements, no modifications will be made to address the original issue concerning the inclusion of the Ownable import in the FrensPoolShare contract.

I07. Presence of Debugging Code in Production Contract

The FrensPoolShare contract contains a commented-out import statement for *hardhat/console.sol*, indicating the presence of debugging code in the production version. This import is commonly used during development for console logging but should be removed in the final production-ready contract.

While the commented-out line does not directly impact the contract's functionality or security, it suggests that the contract may not have undergone thorough cleanup or final review before deployment.

Paths: ./contracts/FrensPoolShare.sol

Recommendation: Eliminate the commented-out import statement from the production version of the contract to maintain code cleanliness.

Found in: 6b73ad4

Status: Fixed (Revised commit: b7b027e)

Remediation: The commented-out debugging import statement was removed.

I08. Inefficient Function Visibility in FrensPoolShare and StakingPool Contracts

Functions that are meant to be exclusively invoked from external sources should be designated as *external* rather than *public*.

Several functions in the FrensPoolShare and StakingPool contracts are declared as *public* when their intended use suggests that *external* visibility would be more appropriate.

Paths: ./contracts/FrensPoolShare.sol : mint(), exists(), renderTokenById(), burn()

./contracts/StakingPool.sol : getIdsInThisPool(), getShare(), getDistributableShare(), getState()

Recommendation: Review each of the affected functions and change their visibility from *public* to *external* if they are only intended to be called externally.

Found in: 6b73ad4

Status: Acknowledged (Revised commit: 413d30f)

Remediation: The visibility of certain functions was changed from public to external in the StakingPool contract, while in the FrensPoolShare contract, the function visibility remains unchanged as originally implemented.

I09. Misalignment Between Documentation and Contract Implementation Regarding exitPool() Function

The documentation for the StakingPool contract references an *exitPool* method, described as a replacement for the former *unstake* function, with a requirement to be called only by FrensOracle. However, the actual contract implementation does not include this *exitPool* function. Instead, the logic to transition the pool's state to *exited* is embedded within the *claim* function.

Code Snippet Indicating Current Implementation:

```
if (currentState != PoolState.exited) {
    IFrensOracle frensOracle =
IFrensOracle(frensStorage.getAddress(keccak256(abi.encodePacked("contract.address",
"FrensOracle"))));
    exited = frensOracle.checkValidatorState(address(this));
    if (exited && currentState == PoolState.staked ){
        currentState = PoolState.exited;
    }
} else exited = true;
```

The inconsistency between the contract's documentation and its actual implementation can lead to confusion among users. Users relying on the

documentation might expect a separate *exitPool* function and may not be aware that the pool's exit logic is integrated into the claim function.

Path: ./contracts/StakingPool.sol

Recommendation:

- Update Documentation to Reflect Current Implementation:
    - Revise the StakingPool contract's documentation to accurately describe the current functionality. Specifically, clarify that the pool's transition to the *exited* state is managed within the *claim* function and not via a separate *exitPool* function.
- Consistency Checks Between Documentation and Code:
    - Conduct a thorough review of both the contract's implementation and its associated documentation to ensure consistency and accuracy across all descriptions and functionalities.

Found in: 6b73ad4

Status: Acknowledged (Revised commit: 413d30f)

Remediation: The revised documentation reflecting the current implementation of the StakingPool contract, particularly the transition to the *exited* state within the claim function, was not shared.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope details

Repository	https://github.com/frens-pool/frens-contracts-v2/
Commit	6b73ad4
Whitepaper	N/A
Requirements	FRENS audit briefing V0.3.pdf
Technical Requirements	FRENS audit briefing V0.3.pdf, NatSpec

Contracts in Scope

./contracts/FrensPoolShare.sol
./contracts/StakingPool.sol
./contracts/interfaces/IDepositContract.sol
./contracts/interfaces/IFrensPoolShare.sol
./contracts/interfaces/IStakingPool.sol
./contracts/interfaces/IFrensArt.sol
./contracts/interfaces/IFrensOracle.sol
./contracts/interfaces/IFrensStorage.sol
./contracts/interfaces/IFrensPoolShareTokenURI.sol