SMART CONTRACT AUDIT REPORT

for

Bedrock Staking

Prepared By: Xiaomi Huang

PeckShield

November 21, 2025

Document Properties

Client	Bedrock
Title	Smart Contract Audit Report
Target	Bedrock Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 21, 2025	Xuxian Jiang	Final Release
1.0-rc1	November 21, 2025	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Bedrock Staking` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Bedrock Staking

`Bedrock` is a blockchain fin-tech company that helps our customers embrace `Web 3.0` effortlessly through the development of innovative products and infrastructure. It also strives to enable institutions and disruptors in the financial and Internet sectors to gain seamless access to blockchain data, crypto yield products and best-in-class key management solutions in a sustainable way. This audit covers the staking support for `ETH 2.0` in allowing users to deposit any number of `Ethers` to the staking contract, and get back equivalent value of `xETH` token (decided by real-time exchange ratio). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Bedrock Staking Protocol

Item	Description
Name	Bedrock
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 21, 2025

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit.

- https://github.com/RockX-SG/stake.git (1b82d38)

And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/RockX-SG/stake.git (03773f6)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Bedrock Staking` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key Bedrock Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Gas Efficiency in Staking::_-stakeInternal()	Coding Practices	Resolved
PVE-002	Low	Timely Settlement Before Applying New Manager Fee Share	Business Logic	Confirmed
PVE-003	Low	Improved Precision in XETH Redemption	Numeric Errors	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Gas Efficiency in Staking::_stakeInternal()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The core `Staking` contract allows users to engage native staking by depositing their `Ethers`. While examining the related staking logic, we notice a minor issue that can be resolved for improved gas efficiency.

In the following, we show the implementation of the related `_stakeInternal()` helper. It is defined as an internal routine to handle the actual staking logic. Specifically, this helper function has a variable named `staked` to keep track of the staked amount for current validator. Each validator theoretically manages up to `DEPOSIT_PER_VALIDATOR_SIZE Ethers` and each `DEPOSIT_SIZE` increment will increase `staked` by `DEPOSIT_SIZE`. In other words, it is better updated as `staked += DEPOSIT_SIZE`, not current `staked = cred.totalStaked + cred.totalReward - cred.totalDebt` (line 431).

```
404     function _stakeInternal() internal {
405         if (totalPending / DEPOSIT_SIZE == 0) {
406             return;
407         }
408         for (uint256 i = 0; i < validatorRegistry.length; i++) {
409             ValidatorCredential storage cred = validatorRegistry[i];
410             if (cred.stopped) {
411                 continue;
412             }
413             // check if we can stake on this validator
414             uint256 staked = cred.totalStaked + cred.totalReward - cred.totalDebt;
415             while (staked < DEPOSIT_PER_VALIDATOR_SIZE && totalPending / DEPOSIT_SIZE >
                    0) {
```

```
                if (!cred.restaking) {
                    _stake(cred.pubkey, cred.signature, withdrawalCredentials);
                } else {
                    address eigenPod = IRestaking(restakingContract).getPod(cred.
                        eigenpod);
                    bytes memory eigenPodCred = abi.encodePacked(bytes1(0x02), new bytes
                        (11), eigenPod);
                    bytes32 restakingWithdrawalCredentials = BytesLib.toBytes32(
                        eigenPodCred, 0);

                    _stake(cred.pubkey, cred.signature, restakingWithdrawalCredentials);
                }

                // track total staked & total pending ethers
                totalStaked += DEPOSIT_SIZE;
                reportedAddedStake += DEPOSIT_SIZE;
                totalPending -= DEPOSIT_SIZE;
                cred.totalStaked += DEPOSIT_SIZE;
                staked = cred.totalStaked + cred.totalReward - cred.totalDebt;
            }
            emit ValidatorStaked(i, staked);
        }
    }
```

Listing 3.1: `Staking::_stakeInternal()`

Further, the core `Staking` contract has another `_dequeueDebt()` function to remove a debt entry from the `etherDebts` array. Our analysis shows it currently returns the dequeued `Debt`, which is not being used. As a result, we can simplify its logic by not having any return value.

Recommendation Improve the above-mentioned routines for improved gas efficiency.

Status This issue has been fixed by the following commit: `03773f6`.

3.2 Timely Settlement Before Applying New Manager Fee Share

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Bedrock Staking` protocol is no exception. Specifically, if we examine the `Staking` contract, it has defined a number of protocol-wide risk parameters, such as managerFeeShare. Our analysis shows that there is a need to timely settle and update the exchange rate before the new manager fee can be applied.

To elaborate, we show below the implementation of the related setter function - setManagerFeeShare (). This risk parameter is used when distributing the rewards into two components, i.e., accountedManagerRevenue and accountedUserRevenue. With that, we need to timely settle latest rewards and update the exchange rate.

```
    function setManagerFeeShare(uint256 milli) external onlyRole(DEFAULT_ADMIN_ROLE) {
        _require(milli >= 0 && milli <= 1000, "SYS008");
        managerFeeShare = milli;

        emit ManagerFeeSet(milli);
    }
```

Listing 3.2: Staking::setManagerFeeShare()

Recommendation Revisit the above logic to timely settle before applying the new managerFeeShare risk parameter.

Status This issue has been confirmed.

3.3 Improved Precision in XETH Redemption

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Staking
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the Staking::redeemFromValidators() as an example. This routine is used to redeem staked funds by turning off associated validators.

```
    function redeemFromValidators(uint256 ethersToRedeem, uint256 maxToBurn, uint256
        deadline)
```

```
963            external
964            nonReentrant
965            returns (uint256 burned)
966        {
967            _require(block.timestamp < deadline, "USR001");
968            _require(ethersToRedeem % DEPOSIT_SIZE == 0, "USR005");
969            _require(ethersToRedeem > 0, "USR005");

971            uint256 totalXETH = IERC20(xETHAddress).totalSupply();
972            uint256 xETHToBurn = totalXETH * ethersToRedeem / currentReserve();
973            _require(xETHToBurn <= maxToBurn, "USR004");

975            // NOTE: the following procedure must keep exchangeRatio invariant:
976            // transfer xETH from sender & burn
977            // uint256 ratio = _exchangeRatioInternal();            // RATIO GUARD BEGIN
978            IMintableContract(xETHAddress).burnFrom(msg.sender, xETHToBurn);
979            _enqueueDebt(msg.sender, ethersToRedeem); // queue ether debts
980            // assert(ratio == _exchangeRatioInternal());          // RATIO GUARD END

982            // return burned
983            return xETHToBurn;
984        }
```

Listing 3.3: `DebtLocker::redeemFromValidators()`

We notice the calculation of the resulting `xETHToBurn` (line 972) involves mixed multiplication and devision. For improved precision, it is better to calculate the result in favor of the protocol, i.e., `xETHToBurn = (totalXETH * ethersToRedeem - 1)/ currentReserve()+ 1`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In Bedrock Staking, there is a privileged administrative account, i.e., the account with the DEFAULT_ADMIN_ROLE role. The administrative account plays a critical role in governing and regulating the staking-wide operations. It also has the privilege to control or govern the flow of assets within the protocol contracts. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the Staking contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setManagerFeeShare(uint256 milli) external onlyRole(DEFAULT_ADMIN_ROLE) {
        _require(milli >= 0 && milli <= 1000, "SYS008");
        managerFeeShare = milli;

        emit ManagerFeeSet(milli);
    }

    /**
     * @dev set eth deposit contract address
     */
    function setETHDepositContract(address _ethDepositContract) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        ethDepositContract = _ethDepositContract;

        emit DepositContractSet(_ethDepositContract);
    }

    /**
     * @dev set restaking contract address
     */
    function setRestakingContract(address _restakingContract) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        restakingContract = _restakingContract;

        emit RestakingAddressSet(_restakingContract);
    }

    /**
     * @dev set withdraw credential to receive revenue, usually this should be the
         contract itself.
     */
```

```
    function setWithdrawCredential(bytes32 withdrawalCredentials_) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        withdrawalCredentials = withdrawalCredentials_;
        emit WithdrawCredentialSet(withdrawalCredentials);
    }
```

Listing 3.4: Example Privileged Operations in `Staking`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms the use of `Aragon DAO` to use these administrative functions.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Bedrock Staking` protocol, which makes it possible for anyone to access efficient and reliable mining and staking services. The staking contract allows users to deposit any number of `Ethers` to the staking contract of `ETH 2.0`, and get back equivalent value of `xETH` token (decided by real-time exchange ratio). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.