Security Audit Report for NearX Exchange Rate Feed Contract and NearX Aurora Staking Contract

Date: October 13, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.10.19 10:56:48 +08'00'

Contents

Report Manifest

Item	Description
Client	Stader Labs
Target	NearX Exchange Rate Feed Contract
	NearX Aurora Staking Contract

Version History

Version	Date	Description
1.0	October 13, 2022	First Release

About BlockSec The BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust and Solidity
Approach	Semi-automatic and manual verification

The repositories that are audited in this report include the following ones.

Repo Name	Github URL
NearX Exchange Rate Feed	https://github.com/stader-labs/nearx-exchange-rate-feed
NearX Aurora	https://github.com/stader-labs/nearx-aurora

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (Version 1), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
NearX Exchange Rate Feed Contract	Version 1	5cca17305b80876590904cc9e42663df17c01d50
	Version 2	8cc689c32c63f6c493d4a2518f54668f2c6688d2
NearX Aurora Staking Contract	Version 1	5a2e9e9ff82b85151104b3e0f88ce7f834889817
	Version 2	19974e00abd0fe373d7a0b452cda3edc3c18fbb8

Note that, we did **NOT** audit all the modules in the repositories. The modules covered by this audit report include **nearx-exchange-rate-feed/near/contract/src** folder contract, **nearx-exchange-rate-feed/aurora/contracts** folder contract, and **nearx-aurora/contracts/AuroraStaking.sol** contract.

Specifically, the file covered in this audit include:

+ nearx-exchange-rate-feed/near/contract/src/
  - contract/public.rs
  - contract/upgrade.rs
  - contract/utils.rs
  - contract.rs
  - errors.rs
  - events.rs
  - lib.rs
  - state.rs
+ nearx-exchange-rate-feed/aurora/contracts/
  - AuroraNearXRate.sol
+ nearx-aurora/contracts/
  - AuroraStaking.sol

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

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

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **two** potential issues. We have **eleven** recommendations and **two** notes.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 2
- Recommendations: 11
- Notes: 2

ID	Severity	Description	Category	Status
1	Low	Missed Sanity Check on the Withdrawal wNear	DeFi Security	Fixed
2	Low	Missed Sanity Check on nearXSwapLockPeriod	DeFi Security	Fixed
3	-	Missed Sanity Check in set_owner()	Recommendation	Fixed
4	-	Missed Sanity Check in get_aurora_contract_address()	Recommendation	Fixed
5	-	Missed Sanity Check When Setting Privileged Accounts	Recommendation	Fixed
6	-	Meaningless Event Emission	Recommendation	Fixed
7	-	Lack of Event Emission in call_aurora()	Recommendation	Fixed
8	-	Improper Usage of the Macro #[private]	Recommendation	Fixed
9	-	Potential Centralization Problem (I)	Recommendation	Confirmed
10	-	Potential Centralization Problem (II)	Recommendation	Confirmed
11	-	Check Zero Address in setAuroraNearXRateAddress()	Recommendation	Fixed
12	-	Follow the Check-Effect-Interactions Best Practice	Recommendation	Fixed
13	-	Unused State Variables	Recommendation	Fixed
14	-	Delayed NearX Rate	Notes	Confirmed
15	-	Timely Pushing the NearX Rate	Notes	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Missed Sanity Check on the Withdrawal wNear

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AuroraStaking` contract, `wNearCollectedFees` and `claimedWNear` are used to store the admin fees and the users' unstaked `wNear`, respectively.

```
176    /// @dev Requested exchange NearX for wNear.
177    /// User can call claimSwapNearXForWNear after nearXSwapLockPeriod
178    /// @param _nearXAmount amount of NearX to be requested for swap.
179    function requestSwapNearXForWNear(uint256 _nearXAmount)
180        external
181        nonReentrant
```

```
182         returns (uint256)
183     {
184         uint256 nearXRate = getNearXRate();
185         uint256 wNearAmount = (_nearXAmount * nearXRate) / EXPONENT_24;
186         uint256 feeAmount = (wNearAmount * wNearToNearXFee) / RATE_CONVERTION;
187         wNearAmount -= feeAmount;
188 
189         require(
190             wNear.balanceOf(address(this)) - claimedWNear >= wNearAmount,
191             "Not enough wNEAR in the pool"
192         );
193 
194         nearX.safeTransferFrom(msg.sender, address(this), _nearXAmount);
195         wNearCollectedFees += feeAmount;
196         claimedWNear += wNearAmount;
197         userNearXSwapRequests[msg.sender].push(
198             NearXSwapRequest(
199                 wNearAmount,
200                 block.timestamp,
201                 block.timestamp + nearXSwapLockPeriod
202             )
203         );
204         uint256 idx = userNearXSwapRequests[msg.sender].length - 1;
205         emit RequestSwapNearXForWNear(
206             msg.sender,
207             _nearXAmount,
208             wNearAmount,
209             feeAmount,
210             idx
211         );
212         return idx;
213     }
```

Listing 2.1: nearx-aurora/contracts/AuroraStaking.sol

In this case, the contract has to reserve enough `wNear` (i.e., `claimedWNear`) for users. However, the current implementation allows the admin to withdraw all the `wNear` tokens, resulting in the assets loss of the other users. Meanwhile, there is no check on whether the withdrawn `wNear` is the collected fees, which can be kept by the admin.

```
303     /// @dev Withdraw wNear pool. Locked for Admin role only
304     /// @param _wNearAmount amount of wNear to withdraw
305     function withdrawWNear(uint256 _wNearAmount)
306         external
307         onlyRole(DEFAULT_ADMIN_ROLE)
308         nonReentrant
309     {
310         require(
311             wNear.balanceOf(address(this)) >= _wNearAmount,
312             "Not enough wNEAR in the pool"
313         );
314 
315         if (_wNearAmount >= wNearCollectedFees) {
```

```
316             wNearCollectedFees = 0;
317         } else {
318             wNearCollectedFees -= _wNearAmount;
319         }
320
321         wNear.safeTransfer(msg.sender, _wNearAmount);
322     }
```

Listing 2.2: nearx-aurora/contracts/AuroraStaking.sol

Impact There may be no enough `wNear` for users to claim and the collected fees are mixed with the unstaked `wNear`.

Suggestion Add a function for withdrawing fees only and limit the maximum withdrawal amount in function `withdrawWNear()`.

2.1.2 Missed Sanity Check on nearXSwapLockPeriod

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AuroraStaking` contract, there is a lockup period after users unstake their `NearX` tokens to get `wNear` tokens, which is specified by `nearXSwapLockPeriod`. The `setNearXSwapLockPeriod` checks the upper bound of this variable but misses the lower bound.

```
244     /// @dev Set lock period for withdraw wNear
245     /// @param _hours time that should pass before user can claim wNear after
            requestSwapNearXForWNear
246     function setNearXSwapLockPeriod(uint256 _hours)
247         external
248         onlyRole(OPERATOR_ROLE)
249     {
250         require(_hours <= 720, "_hours must not exceed 720 (1 month)");
251
252         nearXSwapLockPeriod = _hours * 1 hours;
253
254         emit SetNearXSwapLockPeriodEvent(_hours);
255     }
```

Listing 2.3: nearx-aurora/contracts/AuroraStaking.sol

Impact A short `nearXSwapLockPeriod` allows attackers to temporarily make the wNear balance of the pool extremely low, resulting in a potential balance bias problem.

Suggestion Set a reasonable lower bound for `nearXSwapLockPeriod`.

2.2 Additional Recommendation

2.2.1 Missed Sanity Check in set_owner()

Status Fixed in `Version 2`

Introduced by Version 1

Description In the Exchange Rate Feed contract, there are checks to ensure that the NearXPusher.owner_account_id is different from the NearXPusher.operator_account_id in both function new() and function set_operator(). However, there is no such check in the function set_owner().

```
24  #[init]
25  pub fn new(
26      owner_account_id: AccountId,
27      operator_account_id: AccountId,
28      nearx_account_id: AccountId,
29      aurora_contract_id: String,
30  ) -> Self {
31      require!(
32          owner_account_id != operator_account_id,
33          ERROR_OWNER_OPERATOR_CANNOT_BE_SAME
34      );
35
36      Self {
37          owner_account_id,
38          operator_account_id,
39          nearx_account_id,
40          aurora_contract_id: get_aurora_contract_address(&aurora_contract_id),
41          temp_owner: None,
42          temp_operator: None,
43      }
44  }
```

Listing 2.4: nearx-exchange-rate-feed/near/src/contract/public.rs

```
183 #[payable]
184 pub fn set_operator(&mut self, new_operator_id: AccountId) {
185     assert_one_yocto();
186     self.assert_owner_calling();
187
188     require!(
189         new_operator_id != self.owner_account_id,
190         ERROR_OWNER_OPERATOR_CANNOT_BE_SAME
191     );
192
193     self.temp_operator = Some(new_operator_id.clone());
194
195     Event::SetOperator {
196         old_operator_id: self.operator_account_id.clone(),
197         new_operator_id,
198     }
199     .emit();
200 }
```

Listing 2.5: nearx-exchange-rate-feed/near/src/contract/public.rs

```
119 // Owner update methods
120 #[payable]
121 pub fn set_owner(&mut self, new_owner: AccountId) {
```

```
122         assert_one_yocto();
123         self.assert_owner_calling();
124
125         self.temp_owner = Some(new_owner.clone());
126         Event::SetOwner {
127             old_owner: self.owner_account_id.clone(),
128             new_owner,
129         }
130         .emit();
131     }
```

Listing 2.6: nearx-exchange-rate-feed/near/src/contract/public.rs

Suggestion Add the check to ensure that the `new_owner` account is different from the operator account in function `set_owner()`.

2.2.2 Missed Sanity Check in get_aurora_contract_address()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Exchange Rate Feed` contract, function `get_aurora_contract_address()` returns the input parameter `aurora_contract_id` with the first two bytes removed. However, it only verifies that the input is 42 bytes long and starts with `0x`, but does not check whether the last 40 bytes are all hexadecimal characters.

```
22     pub fn get_aurora_contract_address(aurora_contract_id: &String) -> String {
23         require!(aurora_contract_id.len() == 42, ERROR_AURORA_ADDRESS);
24         require!(aurora_contract_id.starts_with("0x"), ERROR_AURORA_ADDRESS);
25         aurora_contract_id[2..].to_string()
26     }
```

Listing 2.7: nearx-exchange-rate-feed/near/contract/src/contract/util.rs

Suggestion It is recommended to invoke the function `hex::decode()` to ensure that the input parameter `aurora_contract_id` is a valid hexadecimal string.

2.2.3 Missed Sanity Check When Setting Privileged Accounts

Status Fixed in `Version 2`

Introduced by `Version 1`

Description `NearXPusher.owner_account_id`, `NearXPusher.operator_account_id`, and `env::current_account_id()` are all privileged accounts for the `Exchange Rate Feed` contract. However, when setting the `owner` and `operator`, there is no check on whether they are different from the `env::current_account_id()` in the functions listed below. This may lead to a centralization problem.

```
24     #[init]
25     pub fn new(
26         owner_account_id: AccountId,
27         operator_account_id: AccountId,
28         nearx_account_id: AccountId,
```

```
29          aurora_contract_id: String,
30      ) -> Self {
31          require!(
32              owner_account_id != operator_account_id,
33              ERROR_OWNER_OPERATOR_CANNOT_BE_SAME
34          );
35
36          Self {
37              owner_account_id,
38              operator_account_id,
39              nearx_account_id,
40              aurora_contract_id: get_aurora_contract_address(&aurora_contract_id),
41              temp_owner: None,
42              temp_operator: None,
43          }
44      }
```

Listing 2.8: nearx-exchange-rate-feed/near/src/contract/public.rs

```
183     #[payable]
184     pub fn set_operator(&mut self, new_operator_id: AccountId) {
185         assert_one_yocto();
186         self.assert_owner_calling();
187
188         require!(
189             new_operator_id != self.owner_account_id,
190             ERROR_OWNER_OPERATOR_CANNOT_BE_SAME
191         );
192
193         self.temp_operator = Some(new_operator_id.clone());
194
195         Event::SetOperator {
196             old_operator_id: self.operator_account_id.clone(),
197             new_operator_id,
198         }
199         .emit();
200     }
```

Listing 2.9: nearx-exchange-rate-feed/near/src/contract/public.rs

```
119     // Owner update methods
120     #[payable]
121     pub fn set_owner(&mut self, new_owner: AccountId) {
122         assert_one_yocto();
123         self.assert_owner_calling();
124
125         self.temp_owner = Some(new_owner.clone());
126         Event::SetOwner {
127             old_owner: self.owner_account_id.clone(),
128             new_owner,
129         }
130         .emit();
131     }
```

Listing 2.10: nearx-exchange-rate-feed/near/src/contract/public.rs

Suggestion Ensure that the `owner` account, `operator` account and the `contract` account are different when setting any of them.

2.2.4 Meaningless Event Emission

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Exchange Rate Feed` contract, the events emitted in function `commit_owner()` and function `commit_operator()` are meaningless. Take the event `CommitOwner` as an example (lines 144 - 147), the `new_owner` and the `caller` are always the same due to the check in lines 138 - 141.

```
133    #[payable]
134    pub fn commit_owner(&mut self) {
135        assert_one_yocto();
136
137        if let Some(temp_owner) = self.temp_owner.clone() {
138            require!(
139                env::predecessor_account_id() == temp_owner,
140                ERROR_UNAUTHORIZED
141            );
142            self.owner_account_id = self.temp_owner.as_ref().unwrap().clone();
143            self.temp_owner = None;
144            Event::CommitOwner {
145                new_owner: self.owner_account_id.clone(),
146                caller: env::predecessor_account_id(),
147            }
148            .emit();
149        } else {
150            panic!("{}", ERROR_TEMP_OWNER_NOT_SET);
151        }
152    }
```

Listing 2.11: nearx-exchange-rate-feed/near/contract/src/contract/public.rs

```
202    #[payable]
203    pub fn commit_operator(&mut self) {
204        assert_one_yocto();
205
206        if let Some(temp_operator) = self.temp_operator.clone() {
207            require!(
208                env::predecessor_account_id() == temp_operator,
209                ERROR_UNAUTHORIZED
210            );
211            self.operator_account_id = temp_operator;
212            self.temp_operator = None;
213
214            Event::CommitOperator {
215                new_operator_id: self.operator_account_id.clone(),
```

```
216                 caller: env::predecessor_account_id(),
217             }
218             .emit();
219         } else {
220             require!(false, ERROR_TEMP_OPERATOR_NOT_SET);
221         }
222     }
```

Listing 2.12: nearx-exchange-rate-feed/near/contract/src/contract/public.rs

Suggestion It is recommended to emit meaningful events in the above functions.

2.2.5 Lack of Event Emission in call_aurora()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Meaningful events are an important part of smart contract design as they can greatly expose the runtime statistics and support the off-chain analysis. In the `Exchange Rate Feed` contract, there is no such event emitted in function `call_aurora()` to record the rate of `NearX`.

```
87  #[private]
88  pub fn call_aurora(&self, price: u128) {
89      require!(env::promise_results_count() == 1);
90
91      let mut aurora_contract_id = [0u8; 20];
92      hex::decode_to_slice(&self.aurora_contract_id, &mut aurora_contract_id).unwrap();
93
94      let aurora_set_rate_function = NEAR_SET_RATE_FUNCTION_STR;
95      let data: Vec<u8> = aurora_set_rate_function
96          .into_iter()
97          .chain(vec![0u8; 16])
98          .chain(price.to_be_bytes())
99          .collect();
100
101     let input = FunctionCallArgsV1 {
102         contract: aurora_contract_id,
103         input: data,
104     }
105     .try_to_vec()
106     .unwrap();
107
108     let promise0 = env::promise_create(
109         "aurora".parse().unwrap(),
110         "call",
111         input.as_ref(),
112         0,
113         SINGLE_CALL_GAS,
114     );
115
116     env::promise_return(promise0);
117 }
```

Listing 2.13: nearx-exchange-rate-feed/near/contract/src/contract/public.rs

Suggestion It's recommended to emit an event in function `call_aurora()` to record the rate of `NearX` each time it is pushed to `Aurora`.

2.2.6 Improper Usage of the Macro #[private]

Status Fixed in `Version 2`

Introduced by `Version 1`

Description For the `Exchange Rate Feed` contract, functions decorated with macro `#[private]` are usually the callbacks of cross-contract invocations, which means that these functions should only be called by the contract itself. However, the internal function `call_aurora()` that is not a callback is decorated with the `#[private]` macro, which is improper.

```
87    #[private]
88    pub fn call_aurora(&self, price: u128) {
89        require!(env::promise_results_count() == 1);
90
91        let mut aurora_contract_id = [0u8; 20];
92        hex::decode_to_slice(&self.aurora_contract_id, &mut aurora_contract_id).unwrap();
93
94        let aurora_set_rate_function = NEAR_SET_RATE_FUNCTION_STR;
95        let data: Vec<u8> = aurora_set_rate_function
96            .into_iter()
97            .chain(vec![0u8; 16])
98            .chain(price.to_be_bytes())
99            .collect();
100
101       let input = FunctionCallArgsV1 {
102           contract: aurora_contract_id,
103           input: data,
104       }
105       .try_to_vec()
106       .unwrap();
107
108       let promise0 = env::promise_create(
109           "aurora".parse().unwrap(),
110           "call",
111           input.as_ref(),
112           0,
113           SINGLE_CALL_GAS,
114       );
115
116       env::promise_return(promise0);
117   }
```

Listing 2.14: nearx-exchange-rate-feed/near/contract/src/contract/public.rs

Suggestion It is recommended to remove the macro `#[private]` and the function-visibility-modifier `pub` to make the function `call_aurora()` internal.

2.2.7 Potential Centralization Problem (I)

Status Confirmed

Introduced by Version 1

Description The Exchange Rate Feed contract has potential centralization problems. The NearXPusher.owner_account_id has the privilege to set the external contract addresses that interact with this contract (i.e., nearx_account_id and aurora_contract_id), set the privileged account (i.e., operator_account_id), and upgrade the contract.

Suggestion It is recommended to introduce a decentralization design in the contract, such as a a public DAO or multi-signature.

Feedback from the Project owner will be multi-signature.

2.2.8 Potential Centralization Problem (II)

Status Confirmed

Introduced by Version 1

Description The AuroraStaking contract has potential centralization problems. The admin has the privilege to withdraw all the wNear tokens in the contract.

```
303    /// @dev Withdraw wNear pool. Locked for Admin role only
304    /// @param _wNearAmount amount of wNear to withdraw
305    function withdrawWNear(uint256 _wNearAmount)
306        external
307        onlyRole(DEFAULT_ADMIN_ROLE)
308        nonReentrant
309    {
310        require(
311            wNear.balanceOf(address(this)) >= _wNearAmount,
312            "Not enough wNEAR in the pool"
313        );
314
315        if (_wNearAmount >= wNearCollectedFees) {
316            wNearCollectedFees = 0;
317        } else {
318            wNearCollectedFees -= _wNearAmount;
319        }
320
321        wNear.safeTransfer(msg.sender, _wNearAmount);
322    }
```

Listing 2.15: nearx-aurora/contracts/AuroraStaking.sol

Suggestion It is recommended to introduce a decentralization design in the contract, such as a a public DAO or multi-signature.

Feedback from the Project Admin and Manager roles will be multi-sig.

2.2.9 Check Zero Address in setAuroraNearXRateAddress()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AuroraStaking` contract, function `setAuroraNearXRateAddress()` sets the `auroraNearXRateAddress` variable, which is the contract address for retrieving current rates between `NearX` and `wNear` token. However, the `auroraNearXRateAddress` is not checked against zero address.

```
257    /// @dev Set address of the Aurora nearXRate feeding contract. Locked for Operator role only
258    /// @param _auroraNearXRateAddress Address of the Aurora nearXRate feeding contract
259    function setAuroraNearXRateAddress(address _auroraNearXRateAddress)
260        external
261        onlyRole(OPERATOR_ROLE)
262    {
263        emit SetAuroraNearXRateAddress(
264            auroraNearXRateAddress,
265            _auroraNearXRateAddress
266        );
267
268        auroraNearXRateAddress = _auroraNearXRateAddress;
269    }
```

Listing 2.16: nearx-aurora/contracts/AuroraStaking.sol

Suggestion Check whether the `auroraNearXRateAddress` is zero address when it is set.

2.2.10 Follow the Check-Effect-Interactions Best Practice

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `swapWNearForNearX` of contract `AuroraStaking`, the `nearXCollectedFees` is updated after transferring the tokens, which violates the `Check-Effect-Interactions` best practice.

```
152    /// @dev Exchange wNear for NearX
153    /// @param _wNearAmount amount of wNear to be swapped.
154    function swapWNearForNearX(uint256 _wNearAmount) external nonReentrant {
155        uint256 nearXRate = getNearXRate();
156        uint256 nearXAmount = (_wNearAmount * (EXPONENT_24)) / nearXRate;
157        uint256 feeAmount = (nearXAmount * nearXToWNearFee) / RATE_CONVERTION;
158        nearXAmount -= feeAmount;
159
160        require(
161            nearX.balanceOf(address(this)) >= nearXAmount,
162            "Not enough NearX in the pool"
163        );
164
165        wNear.safeTransferFrom(msg.sender, address(this), _wNearAmount);
166        nearX.safeTransfer(msg.sender, nearXAmount);
167        nearXCollectedFees += feeAmount;
168        emit SwapWNearForNearX(
169            msg.sender,
```

```
170             _wNearAmount,
171             nearXAmount,
172             feeAmount
173         );
174     }
```

Listing 2.17: nearx-aurora/contracts/AuroraStaking.sol

Suggestion Refact the code to follow the `Check-Effect-Interactions` best practice.

2.2.11 Unused State Variable

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AuroraStaking` contract, The `decimals` state variable is not used.

Suggestion Remove the unused state variable.

2.3 Notes

2.3.1 Delayed NearX Rate

Status Confirmed

Introduced by `Version 1`

Description Given the async nature of `NEAR` protocol, one transaction on `NEAR` protocol may be executed in several blocks. Therefore, it should be noted that the rate of `NearX` pushed to `Aurora` would not be the latest for the `Exchange Rate Feed` contract.

Feedback from the Project Exchange rate is updated on 2 occasions only in an epoch for autocompounding and boosted rewards addition. We are currently planning to call the `push_nearx_rate_to_aurora` method every 10mins when we launch. This will ensure that the rate is always accurate, since we autocompound once an epoch (rewards are accrued only once an epoch) and boost rewards once a day currently.

2.3.2 Timely Pushing the NearX Rate

Status Confirmed

Introduced by `Version 1`

Description In the `Exchange Rate Feed` contract, function `push_nearx_rate_to_aurora()` is used to push the latest rate of `NearX` from NEAR's mainnet to Aurora. It's important for the team to make sure that the function will be triggered by the operator timely.

Feedback from the Project We will ensure that it is timely triggered by us. Going forward we will consider opening up a method to be public once the product reaches a certain level of stability.