Aurora - Staking Farm

NEAR Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **February 9th, 2022 - March 25th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	02/22/2022	Mustafa Hasan
0.2	Document Edits	03/08/2022	Mustafa Hasan
0.3	Document Edits	03/19/2022	Mustafa Hasan
0.4	Document Edits	03/22/2022	Timur Guvenkaya
0.5	Final Draft	03/25/2022	Timur Guvenkaya
0.6	Draft Review	03/25/2022	Gabi Urrutia
1.0	Remediation Plan	04/20/2022	Mustafa Hasan
1.1	Remediation Plan Review	04/20/2022	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
Mustafa Hasan	Halborn	Mustafa.Hasan@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Aurora engaged Halborn to conduct a security assessment on the staking farm NEAR smart contracts utilized by them, beginning on February 9th, 2022 and ending March 25th, 2022. Aurora provides Ethereum compatibility, NEAR Protocol scalability, and industry-first user experience through affordable transactions.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverables set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure development.

1.2 AUDIT SUMMARY

The team at Halborn was provided 6 weeks for the engagement and assigned two full-time security engineers to audit the security of the assets in scope. The engineers are blockchain and smart contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to achieve the following:

- Identify potential security issues within the NEAR smart contracts.

In summary, Halborn identified few security risks that were mostly addressed by the Aurora team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual view of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts 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, purpose, and use of the platform.
- Manual code read and walkthrough.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Fuzz testing. (cargo fuzz, honggfuzz)
- Checking the unsafe code usage. (cargo-geiger)
- Scanning of Rust files for vulnerabilities.(cargo audit)
- Deployment to devnet through near-cli

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.

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

- Staking Factory
- Staking Farm

EXECUTIVE OVERVIEW

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	4	3

LIKELIHOOD

IMPACT

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

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - PUBLICLY CALLABLE FUNCTIONS LEADING TO OUT-OF-CONTRACT FUNDS BURN	High	NOT APPLICABLE
HAL02 - IMPROPER ROLE-BASED ACCESS CONTROL POLICY	High	PARTIALLY SOLVED
HAL03 - MULTIPLE STAKING ACTIONS CAN BE PERFORMED WHILE CONTRACT IS PAUSED	Medium	SOLVED - 04/12/2022
HAL04 - LACK OF VALIDATION OF BURN FRACTION	Medium	SOLVED - 04/12/2022
HAL05 - VALUE CONVERSION TO SMALLER SIZES MAY RESULT IN OVERFLOWS	Low	SOLVED - 04/12/2022
HAL06 - DELEGATOR AND PREDECESSOR CAN BE THE SAME	Low	NOT APPLICABLE
HAL07 - USE OF VULNERABLE CRATES	Low	RISK ACCEPTED
HAL08 - DEPOSIT ATTACHED IS NOT ASSERTED	Low	NOT APPLICABLE
HAL09 - REDUNDANT ASSERTION	Informational	SOLVED - 04/12/2022
HAL10 - ASSERTION SHOULD BE REPLACED BY A MACRO	Informational	SOLVED - 04/12/2022
HAL11 - DEFAULT IMPLEMENTATION SHOULD BE REPLACED BY A MACRO	Informational	SOLVED - 04/12/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) HAL01 - PUBLICLY CALLABLE FUNCTIONS LEADING TO OUT-OF-CONTRACT FUNDS BURN - HIGH

Description:

The unstake_burn() and burn() functions in “staking-farm/src/stake.rs” can be publicly callable by anyone, allowing malicious users to continually call the functions with each new epoch, which leads to the reduction of the total stakes in the pool, which would result in fewer rewards for each user who stakes and the transfer of all unstaked tokens to address zero.

Code Location:

Listing 1: staking-farm/src/stake.rs

```
pub fn unstake_burn(&mut self) {
    self.internal_unstake_all(&AccountId::new_unchecked(
↳ ZERO_ADDRESS.to_string()));
}
```

Listing 2: staking-farm/src/stake.rs

```
pub fn burn(&mut self) {
    let account_id = AccountId::new_unchecked(ZERO_ADDRESS.
↳ to_string());
    let account = self.internal_get_account(&account_id);
    if account.unstaked > MIN_BURN_AMOUNT {
        // TODO: replace with burn host function when available.
        self.internal_withdraw(&account_id, account.unstaked);
    }
}
```

Proof of Concept:

The following test case was developed to showcase the issue:

Listing 3

```
fn public_token_burning() {
    let (root, pool) = setup(to_yocto("5"), 1, 3);
    let user1 = create_user_no_stake(&root, &pool);
    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));
    wait_epoch(&root);
    //User forces all of the zero address account's tokens to be unstaked
    assert_all_success(call!(user1, pool.unstake_burn()));
    println!("Unstaked balance: {}", to_int(view!(pool.get_account_unstaked_balance(burn_account()))));
    //Wait for epochs before funds can be withdrawn to the zero address account (effectively burning them)
    wait_epoch(&root);
    wait_epoch(&root);
    wait_epoch(&root);
    wait_epoch(&root);
    assert_all_success(call!(user1, pool.burn()));
    //Zero address should have zero unstaked tokens
    println!("Unstaked balance: {}", to_int(view!(pool.get_account_unstaked_balance(burn_account()))));
}
```

Risk Level:

Likelihood - 4
Impact - 5

Recommendation:

Check if the owner is calling the functions before executing their logic, otherwise revert.

Remediation Plan:

NOT APPLICABLE: The team accepts this behavior as it is intentional based on the reasoning at https://github.com/referencedev/staking-farm#burning-rewards

3.2 (HAL-02) HAL02 - IMPROPER ROLE-BASED ACCESS CONTROL POLICY - HIGH

Description:

It was observed that most of the privileged functionality is controlled by the owner. Additional authorization levels are needed to implement the principle of least privilege, also known as least authority, which ensures that only authorized processes, users, or programs can access necessary resources or information. Role ownership is useful in a simple system, but more complex projects require more roles by using role-based access control policy.

Code Location:

The owner can access those functions:

- stop_farm function in farm.rs
- All functions in owner.rs

Risk Level:

Likelihood - 3
Impact - 5

Recommendation:

Adding additional roles is recommended to adhere to the principle of least privilege and limit owner privileges. You can include the pauser role and change assert_owner_or_authorized_user() to allow only authorized users to perform actions. Also, do not allow the owner to be set as an authorized user via add_authorized_user.

Remediation Plan:

PARTIALLY SOLVED: The Aurora team introduced a fix that separates owner and pauser permissions in https://github.com/referencedev/staking-farm/pull/11. However, the fix is partial, since the owner can still become a pauser and the pausers list could become empty by removing all pausers.

3.3 (HAL-03) HAL03 - MULTIPLE STAKING ACTIONS CAN BE PERFORMED WHILE CONTRACT IS PAUSED - MEDIUM

Description:

The internal_restake() function in "staking-farm/src/internal.rs" checks if the contract is paused before performing its internal logic, however multiple functions that perform other staking actions do not perform that check before execution, allowing staking actions to be carried out even when staking is paused. Such functions include internal_stake() and inner_unstake().

Risk Level:

Likelihood - 2
Impact - 5

Recommendation:

All functions that perform logic that affects staking actions should start by checking whether the contract is paused or not.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

3.4 (HAL-04) HAL04 - LACK OF VALIDATION OF BURN FRACTION - MEDIUM

Description:

When a new instance of StakingContract is created, a burn fraction has to be provided and is then used to determine the amount of tokens burned with each call to the ping() function. An assert_valid() function is implemented on the Ratio struct that represents the fraction, however it is never called on the passed fraction value before it is used in the StakingContract. This allows an owner to carry out the following scenarios:

1. Create a staking pool with a burn fraction that evaluates to 1, meaning all rewards will be burned and nothing will remain for the owner and delegators
2. Create a staking pool with a burn fraction that evaluates to more than 1, which will cause a panic case every time internal_ping() is called
3. Create a staking pool with a burn fraction that evaluates to 0, meaning nothing will ever burn, which would allow the owner to basically harvest all the rewards if they set the reward fee to a fraction that evaluates to 1

Code Location:

Listing 4: staking-farm/src/lib.rs (Lines 198,221)

```
193 #[init]
194     pub fn new(
195         owner_id: AccountId,
196         stake_public_key: PublicKey,
197         reward_fee_fraction: Ratio,
198         burn_fee_fraction: Ratio,
199     ) -> Self {
200         assert!(!env::state_exists(), "Already initialized");
201         reward_fee_fraction.assert_valid();
202         assert!(
```

```
            env::is_valid_account_id(owner_id.as_bytes()),
            "The owner account ID is invalid"
        );
        let account_balance = env::account_balance();
        let total_staked_balance = account_balance - STAKE_SHARE_PRICE_GUARANTEE_FUND;
        assert_eq!(
            env::account_locked_balance(),
            0,
            "The staking pool shouldn't be staking at the initialization"
        );
        let mut this = Self {
            stake_public_key: stake_public_key.into(),
            last_epoch_height: env::epoch_height(),
            last_total_balance: account_balance,
            total_staked_balance,
            total_stake_shares: NumStakeShares::from(total_staked_balance),
            total_burn_shares: 0,
            reward_fee_fraction: UpdatableRewardFee::new(reward_fee_fraction),
            burn_fee_fraction,
            accounts: UnorderedMap::new(StorageKeys::Accounts),
            farms: Vector::new(StorageKeys::Farms),
            active_farms: Vec::new(),
            paused: false,
            authorized_users: UnorderedSet::new(StorageKeys::AuthorizedUsers),
            authorized_farm_tokens: UnorderedSet::new(StorageKeys::AuthorizedFarmTokens),
        };
```

Proof of Concept::

Test cases were done and indeed they resulted in 0 rewards, panic and the owner collected the full reward for the 3 cases mentioned above, respectively:

Listing 5: Burning all rewards

```
fn burn_all_rewards() {
    let (root, pool) = setup(to_yocto("10000") + 1_000_000_000_000, 10, 10);
    let _ = create_user_and_stake(&root, &pool);
    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));

    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));
}
```

Listing 6: Panic on every ping() function call

```
fn panic_on_ping() {
    let (root, pool) = setup(to_yocto("10000") + 1_000_000_000_000, 10, 11);
    let _ = create_user_and_stake(&root, &pool);
    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));
}
```

Listing 7: Owner getting all rewards

```
fn owner_gets_all_rewards() {
    let (root, pool) = setup(to_yocto("10000") + 1_000_000_000_000, 10, 0);
    let user1 = create_user_and_stake(&root, &pool);
    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));

    let mut root_balance = to_int(view!(pool.get_account_total_balance(root.account_id())));
    let mut user_balance = to_int(view!(pool.get_account_total_balance(user1.account_id())));

    log!("First iteration: Root balance: {}\nUser balance: {}", root_balance, user_balance);

    wait_epoch(&root);
    assert_all_success(call!(root, pool.ping()));

```

```
    root_balance = to_int(view!(pool.get_account_total_balance(root.account_id())));
    user_balance = to_int(view!(pool.get_account_total_balance(user1.account_id())));

    log!("Second iteration: Root balance: {}\nUser balance: {}", root_balance, user_balance);
}
```

Risk Level:

Likelihood - 2
Impact - 5

Recommendation:

The assert_valid() function must be called before the fraction is used to create the StakingContract instance.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

3.5 (HAL-05) HAL05 - VALUE CONVERSION TO SMALLER SIZES MAY RESULT IN OVERFLOWS - LOW

Description:

This behavior exists in multiple areas of the project, for example in the multiply() function implemented for the Ratio struct in “staking-farm/src/lib.rs”. It is required to enforce that the ratio is valid.

Code Location:

Listing 8: staking-farm/src/lib.rs

```
pub fn multiply(&self, value: Balance) -> Balance {
        if self.denominator == 0 || self.numerator == 0 {
            0
        } else {
            (U256::from(self.numerator) * U256::from(value) / U256::from(self.denominator))
                .as_u128()
        }
    }
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Ratio validation should always take place to avoid cases of overflow.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

3.6 (HAL-06) HAL06 - DELEGATOR AND PREDECESSOR CAN BE THE SAME - LOW

Description:

It was observed that the `claim()` function accepts that `delegator_id` is equal to `env::predecessor_account_id()`. Enabling this will cause the smart contract to perform a redundant operation of doing a cross contract call to the delegator and then setting `claim_account_id` and `send_account_id` to the same value in `internal_claim()`.

Code Location:

- staking-farm/src/farm.rs: claim()

Recommendation:

Consider asserting `delegator_id != env::predecessor_account_id()` to avoid redundant operations.

Remediation Plan:

NOT APPLICABLE: The Aurora team will not fix since it does not pose a direct risk and updating the code might introduce other bugs.

3.7 (HAL-07) HAL07 - USE OF VULNERABLE CRATES - LOW

Description:

The following crates used in the project dependencies have known vulnerabilities:

ID	package	Short Description
RUSTSEC-2020-0159	chrono	Potential segfault in ‘localtime_r‘ invocations
RUSTSEC-2021-0067	cranelift-codegen	Memory access due to code generation flaw in Cranelift module
RUSTSEC-2021-0013	raw-cpuid	Soundness issues in ‘raw-cpuid‘
RUSTSEC-2021-0089	raw-cpuid	Optional ‘Deserialize‘ implementations lacking validation
RUSTSEC-2022-0013	regex	Regexes with large repetitions on empty sub-expressions take a very long time to parse
RUSTSEC-2020-0071	time	Potential segfault in the time crate
RUSTSEC-2021-0110	wasmtime	Multiple Vulnerabilities in Wasmtime

3.8 Recommendation

Even if those vulnerable crates cannot affect the underlying application, it is recommended to be aware of them. Furthermore, you need to configure dependency monitoring to always be alert when a new vulnerability is disclosed in one of the project crates.

Remediation Plan:

RISK ACCEPTED: The Aurora team accepted the risk of this finding; however, no fixes were introduced as the affected crates are not under the team’s control.

3.9 (HAL-08) HAL08 - DEPOSIT ATTACHED IS NOT ASSERTED - LOW

Description:

The deposit() function does not assert that the attached deposit works. Users can call this function without attaching a deposit by making the amount zero in the internal_deposit function.

Code Location:

- staking-farm/src/stake.rs: deposit()

Recommendation:

It is advised to assert at least one to avoid any redundant calls to that function.

NOT APPLICABLE: The Aurora team decided this will not be fixed since it does not pose a direct risk.

3.10 (HAL-09) HAL09 - REDUNDANT ASSERTION - INFORMATIONAL

Description:

In the new function, an assert prevents anyone from re-initializing the contract. However, since the #[init] macro is used, this check is redundant.

Code Location:

Listing 9: staking-farm/src/lib.rs (Line 200)

```
193 #[init]
194     pub fn new(
195         owner_id: AccountId,
196         stake_public_key: PublicKey,
197         reward_fee_fraction: Ratio,
198         burn_fee_fraction: Ratio,
199     ) -> Self {
200         assert!(!env::state_exists(), "Already initialized");
201         reward_fee_fraction.assert_valid();
202         ...
```

Recommendation:

Consider removing that assertion to avoid redundant code.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

3.11 (HAL-10) HAL10 - ASSERTION SHOULD BE REPLACED BY A MACRO - INFORMATIONAL

Description:

In the on_stake_action function, the assert statement is used to ensure that the function is only callable by the contract itself. However, near_sdk already provides the #[private] macro, which can be used to do that.

Code Location:

Listing 10: staking-farm/src/stake.rs (Line 146)

```
145 pub fn on_stake_action(&mut self) {
146         assert_eq!(
147             env::current_account_id(),
148             env::predecessor_account_id(),
149             "Can be called only as a callback"
150         );
151 ...
152 }
```

Recommendation:

Consider adding the #[private] macro which implements the same check.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

3.12 (HAL-11) HAL11 - DEFAULT IMPLEMENTATION SHOULD BE REPLACED BY A MACRO - INFORMATIONAL

Description:

The /staking-farm/staking-farm/src/lib.rs contract contains a default implementation of a contract that triggers the assertion. However, instead of coding it yourself, there is a macro called PanicOnDefault that you can bypass.

Code Location:

Listing 11: staking-farm/src/lib.rs

```
impl Default for StakingContract {
    fn default() -> Self {
        panic!("Staking contract should be initialized before
 ↳ usage")
    }
}
```

Recommendation:

Consider bypassing PanicOnDefault to remove that default implementation.

Remediation Plan:

SOLVED: The Aurora team fixed this issue in https://github.com/referencedev/staking-farm/pull/11.

AUTOMATED TESTING

4.1 AUTOMATED ANALYSIS

Description:

Halborn used automated security scanners to assist with 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-2020-0159	chrono	Potential segfault in ‘localtime_r‘ invocations
RUSTSEC-2021-0067	cranelift-codegen	Memory access due to code generation flaw in Cranelift module
RUSTSEC-2021-0013	raw-cpuid	Soundness issues in ‘raw-cpuid‘
RUSTSEC-2021-0089	raw-cpuid	Optional ‘Deserialize‘ implementations lacking validation
RUSTSEC-2022-0013	regex	Regexes with large repetitions on empty sub-expressions take a very long time to parse
RUSTSEC-2020-0071	time	Potential segfault in the time crate
RUSTSEC-2021-0110	wasmtime	Multiple Vulnerabilities in Wasmtime

THANK YOU FOR CHOOSING

HALBORN