MetaPool - Katherine Fundraising and Bond Market

NEAR Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **April 10th, 2023 - May 12th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	05/11/2023	Michal Bajor
0.2	Document Updates	05/11/2023	Michal Bajor
0.3	Draft Version	05/11/2023	Michal Bajor
0.4	Draft Review	05/12/2023	Alpcan Onaran
0.5	Draft Review	05/13/2023	Gabi Urrutia
1.0	Remediation Plan	06/07/2023	Michal Bajor
1.1	Remediation Plan Review	06/07/2023	Alpcan 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
Alpcan Onaran	Halborn	Alpcan.Onaran@halborn.com
Michal Bajor	Halborn	Michal.Bajor@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MetaPool engaged the services of Halborn to execute a security audit on their smart contracts. This examination of their systems transpired over a month-long period, commencing on the 10th of April 2023, and culminating on the 12th of May 2023.

The scope of the security assessment was defined to encompass only the smart contracts which had been provided in the katherine-fundraising repository. The exact versions of the smart contracts that were included in this assessment were determined by certain commit hashes, the specifics of which are elaborately laid out in the Scope section of this report. This level of detail ensures clarity about the exact content and versions of the contracts that were scrutinized during the audit.

The client's project, Katherine, is a sophisticated crowd fundraising initiative. Its unique selling proposition lies in its ingenious leveraging of the yield produced by staking NEAR tokens in the Meta Pool. This method allows for the generation of funds in a decentralized, secure, and transparent manner.

The system's design incorporates the Bond Market and Bond Operator contracts, which operate in tandem to create a functional and efficient market for bonds associated with funding. This bond market serves as a crucial element of the Katherine project, providing an additional layer of financial sophistication and flexibility. It allows the contributors to the project to not only support a cause they believe in, but also potentially reap a financial return on their investment.

1.2 AUDIT SUMMARY

The team at Halborn was provided 4 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
- Ensure all functions are running as intended

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which has been successfully addressed by MetaPool . The main ones are the following:

FUNDS LOCKING DUE TO UNAUTHORIZED BOND MERGING

It was observed that anyone could merge bonds that are valid for such an operation. Although this action does not change the underlying value, the issue originates from the fact that one of the bonds might have been put on sale prior to merging. In such a scenario, if the auction sale was successful, however, a malicious user will merge that bond into a different one, it will be impossible to change its ownership and the funds sent as an auction bid will be locked.

MetaPool **successfully** remediated this issue by introducing a verification mechanism that prevents merging bonds that are on sale. Additionally, only the owner of both bonds can merge them.

LOSS OF REWARDS DUE TO KICKSTARTER UPDATE

It was observed that the kick-starter could be updated prior to its funding time frame. Doing so results in zeroing the variable responsible for representing project tokens that were sent as a reward for supporters. If some project tokens were sent as rewards to supporters, they will effectively be lost even if the AccountId associated with the token contract remained the same.

MetaPool **successfully** remediated this issue by deprecating the kickstarter update function.

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, workspaces-rs)
- 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:

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

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. Katherine Fundraising

- Repository: katherine-fundraising
- Commit ID: bfc38054194ad37c531c532645edfd5a7bde3933
- Smart Contracts in scope:
    1. Katherine Fundraising (katherine-fundraising/contracts/katherine-fundraising-contract/)
    2. Bond Operator (katherine-fundraising/contracts/bond-operator-contract)
    3. Bond Market (katherine-fundraising/contracts/bond-market-contract)

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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	1	9

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FUNDS LOCKING DUE TO UNAUTHORIZED BOND MERGING	Critical (9.4)	SOLVED - 05/06/2023
LOSS OF REWARDS DUE TO KICKSTARTER UPDATE	Low (2.0)	SOLVED - 05/17/2023
DENIAL OF SERVICE CONDITION DUE TO STORAGE BLOATING	Informational (0.5)	SOLVED - 06/06/2023
REDUNDANT STATE VALIDATION	Informational (0.0)	SOLVED - 05/17/2023
REDUNDANT MANUAL CALLBACK ASSERTION	Informational (0.0)	SOLVED - 05/18/2023
NOT NECESSARY MACRO USAGE	Informational (0.0)	SOLVED - 05/18/2023
REDUNDANT FUNCTION	Informational (0.0)	SOLVED - 05/18/2023
DEAD CODE	Informational (0.0)	SOLVED - 05/17/2023
JAVASCRIPT INCOMPATIBLE TYPE	Informational (0.0)	SOLVED - 05/18/2023
POSSIBLE OPTIMIZATIONS TO REDUCE BINARY SIZE	Informational (0.0)	SOLVED - 05/18/2023
OUTDATED DEPENDENCIES	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) FUNDS LOCKING DUE TO UNAUTHORIZED BOND MERGING - CRITICAL(9.4)

Description:

During our analysis, we identified an issue concerning the merge_bonds function within the system. This function currently lacks any form of authorization, which leaves it open to manipulation by any user, including those with malicious intent. This absence of secure access control allows users to merge any valid bonds, despite the possible implications to the system's stability and security.

One such implication is the potential for funds to be indefinitely locked within the BondMarket contract. This problematic scenario occurs when a bond, which is listed for auction sale within the BondMarket contract, is successfully auctioned off. In this case, the winning bidder should ideally be able to claim ownership of the bond via the pull_sale_bond function, which transfers ownership through a cross-contract call to the BondOperator contract.

However, should this cross-contract call fail for any reason, the system reverts to a state where the winning bidder is still recognized as the auction winner through the process_auction_ends_callback function. This state of affairs leads to the funds remaining inaccessible for withdrawal as long as the system perceives that user as the auction winner.

The unrestricted access to the merge_bonds function adds another layer of complexity to this issue. A malicious user could manipulate this function, merging a bond currently on sale into another bond, effectively erasing the original bond's identifier data. The deletion of this crucial data in turn results in the persistent failure of the cross-contract call initiated by the pull_sale_bond function.

This vulnerability effectively causes the funds committed by the winning bidder to be indefinitely locked within the BondMarket contract.

Moreover, it obstructs the proper transfer of bond ownership, creating potential liabilities and hindering the smooth operation of the system.

Code Location:

Down below is a code snippet from the merge_bonds function:

Listing 1: contracts/bond-operator-contract/src/lib.rs

```
pub fn merge_bonds(&mut self, bond_id: BondId, other_id: BondId) {
    let bond = self.internal_get_bond(bond_id);
    let mut other = self.internal_get_bond(other_id);
    let new_bond = bond.merge(&mut other);

    // Remove other bond.
    self.internal_bond_drop(&other);

    // Replace with merged bond.
    self.bonds.insert(&bond_id, &new_bond);
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:C (9.4)

Proof Of Concept:

Listing 2: src/halborn_testcases/test_merge_bond_on_sale.rs

```
#[tokio::test]
async fn test_merge_bond_on_sale() -> anyhow::Result<()> {
    let worker = workspaces::sandbox().await?;

    let root_account = worker.root_account()?;

    let halborn_account = root_account
        .create_subaccount("halborn")
        .initial_balance(1999999999999999000000000000)
        .transact()
        .await?
```

```
            .into_result()?;

    let katherine_owner_account = root_account
        .create_subaccount("katowner")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let kickstarter_owner_account = root_account
        .create_subaccount("kickowner")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let metapool_account = root_account
        .create_subaccount("metapool")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let katherine_account = root_account
        .create_subaccount("katherine")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let supporter_account = root_account
        .create_subaccount("supporter")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let buyer_account = root_account
        .create_subaccount("buyer")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

```

```
    let bond_market_owner_account = root_account
        .create_subaccount("bondmarket-owner")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let bond_operator_owner = root_account
        .create_subaccount("bondoperator-owner")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let bond_market_operator_account = root_account
        .create_subaccount("bondoperator")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let bond_market_account = root_account
        .create_subaccount("bondmarket")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    let ptoken_account = root_account
        .create_subaccount("ptoken")
        .initial_balance(199999999999999900000000000)
        .transact()
        .await?
        .into_result()?;

    // deploying contracts
    let metapool_contract = deploy_meta_pool(&katherine_owner_account, &metapool_account).await?;

    let ptoken_contract = deploy_ptoken(&katherine_owner_account, &ptoken_account).await?;
    let bond_market_contract = deploy_bond_market(
        &bond_market_owner_account,
        &bond_market_account,
```

```
        metapool_contract.id(),
    )
    .await?;
    let bond_operator_contract = deploy_operator(
        &bond_operator_owner,
        &bond_market_operator_account,
        &metapool_account,
        &bond_market_contract,
        &bond_market_owner_account,
    )
    .await?;

    let current_epoch: EpochMillis = metapool_contract
        .call("get_epoch")
        .args_json(json!({}))
        .view()
        .await?
        .json()?;
    println!("CURRENT EPOCH: {}", current_epoch);

    mint_ptokens(
        &ptoken_contract,
        &katherine_owner_account,
        vec![
            //&katherine_owner_account,
            &katherine_account,
            &supporter_account,
            &buyer_account,
        ],
    )
    .await?;

    let now = Now::new_from_epoch_millis(metapool_contract.call("get_epoch").view().await?.json()?);

    let stnear_freeze_timestamp: EpochMillis = now.to_epoch_millis();
    let stnear_vault_maturity_datetime: EpochMillis = now.increment_min(40).to_epoch_millis();
    let ptoken_start_linear_release_datetime: EpochMillis = now.increment_min(15).to_epoch_millis();
    let ptoken_vault_maturity_datetime: EpochMillis = now.increment_min(40).to_epoch_millis();

```

```
    // Import testing bonds
    let content = fs::read_to_string(BONDS_FILEPATH).expect("Error
 ↳ reading bond file");
    let bonds_json = json_reader::parse(&content)?;
    let bonds = bonds_json["bonds"].clone();

    // Vault parameters
    let vault_id = String::from("TEST_vault_id");
    let stnear_price_at_freeze: U128 = metapool_contract
        .call("get_st_near_price")
        .view()
        .await?
        .json()?;
    let initial_stnear_balance: U128 =
        calculate_stnear_balance(bonds.clone(),
 ↳ stnear_price_at_freeze.clone());
    let initial_ptoken_balance: U128 = calculate_ptoken_balance(
 ↳ bonds.clone());
    let ptoken_contract_address: AccountId = ptoken_contract.id().
 ↳ clone();
    let interest_beneficiary_until_unfreeze: AccountId =
 ↳ kickstarter_owner_account.id().clone();
    let interest_beneficiary_near_claimed: U128 = U128::from(0);
    let bond_owners_near_claimed: U128 = calculate_near_claimed(
 ↳ bonds.clone());
    let bond_owners_ptoken_claimed: U128 =
 ↳ calculate_ptoken_claimed(bonds.clone());
    let vault_owner_id: AccountId = katherine_owner_account.id().
 ↳ clone();

    // Create a vault
    let res = bond_operator_owner
        .call(bond_operator_contract.id(), "create_vault")
        .args_json(serde_json::json!({
           "vault_id": vault_id,
            "stnear_price_at_freeze": stnear_price_at_freeze,
            "initial_stnear_balance": initial_stnear_balance,
            "initial_ptoken_balance": initial_ptoken_balance,
            "ptoken_contract_address": ptoken_contract_address,
            "stnear_freeze_timestamp": U64::from(
 ↳ stnear_freeze_timestamp),
            "interest_beneficiary_until_unfreeze":
 ↳ interest_beneficiary_until_unfreeze,
```

```
            "interest_beneficiary_near_claimed": interest_beneficiary_near_claimed,
            "bond_owners_near_claimed": bond_owners_near_claimed,
            "bond_owners_ptoken_claimed": bond_owners_ptoken_claimed,
            "stnear_vault_maturity_datetime": U64::from(stnear_vault_maturity_datetime),
            "ptoken_start_linear_release_datetime": U64::from(ptoken_start_linear_release_datetime),
            "ptoken_vault_maturity_datetime": U64::from(ptoken_vault_maturity_datetime),
            "vault_owner_id": vault_owner_id,
        }))
        .gas(parse_gas!("200 Tgas") as u64)
        .transact()
        .await?
        .into_result()?;
    println!("Create vault: {:?}\n", res);

    registering_accounts(
        &metapool_contract,
        &ptoken_contract,
        &bond_operator_contract,
        &katherine_owner_account,
        &supporter_account,
        &buyer_account,
        &kickstarter_owner_account,
        &bond_operator_owner,
    )
    .await?;

    sending_stnear_ptoken_to_vault(
        &metapool_contract,
        &ptoken_contract,
        &bond_operator_contract,
        &katherine_owner_account,
        &bond_operator_owner,
        initial_stnear_balance,
        initial_ptoken_balance,
        vault_id.clone(),
    )
    .await?;
    sending_stnear_ptoken_to_vault(
        &metapool_contract,
```

```
        &ptoken_contract,
        &bond_operator_contract,
        &katherine_owner_account,
        &bond_operator_owner,
        initial_stnear_balance,
        initial_ptoken_balance,
        vault_id.clone(),
    )
    .await?;

    let loader_bonds = create_bond_loader(
        bonds,
        vault_id.clone(),
        supporter_account.id().clone(),
        ptoken_contract_address.clone(),
        U64::from(4 * stnear_vault_maturity_datetime),
        U64::from(4 * ptoken_start_linear_release_datetime),
        U64::from(4 * ptoken_vault_maturity_datetime),
    );

    let res = bond_operator_owner
        .call(bond_operator_contract.id(), "create_bonds")
        .args_json(serde_json::json!({ "bonds": loader_bonds }))
        .gas(parse_gas!("200 Tgas") as u64)
        .transact()
        .await?
        .into_result()?;
    println!("Create bonds: {:?}\n", res);

    let bond1: BondJSON = bond_operator_contract
        .call("get_bond")
        .args_json(json!({
            "bond_id": 1,
        }))
        .view()
        .await?
        .json()?;
    println!("\nBOND1: {:#?}", bond1);

    let bond3: BondJSON = bond_operator_contract
        .call("get_bond")
        .args_json(json!({
            "bond_id": 3,
        }))
```

```
        .view()
        .await?
        .json()?;
    println!("\nBOND3: {:#?}", bond3);

    // 1. Putting bond 3 on sale
    let price = U128::from(10 * ONE_NEAR);
    let bond_sale_result = supporter_account
        .call(bond_market_contract.id(), "create_bond_sale_in_millisecs")
        .args_json(json!({
            "bond_id": 3,
            "bond_operator_address": bond_operator_contract.id(),
            "price": price,
            "currency": "NEAR",
            "sale_duration_in_millisecs": U64::from(20 * 1000), // 20 seconds
            "is_auction": true,
        }))
        .max_gas()
        .deposit(1000000000000000000000000)
        .transact()
        .await?
        .into_result()?;
    println!("SALE RESULT: {:#?}", bond_sale_result);

    let bond3: BondJSON = bond_operator_contract
        .call("get_bond")
        .args_json(json!({
            "bond_id": 3,
        }))
        .view()
        .await?
        .json()?;
    println!("\nBOND3 after putting on sale: {:#?}", bond3);

    let sale: SaleJSON = bond_market_contract
        .call("get_sale")
        .args_json(json!({"sale_id": 0}))
        .view()
        .await?
        .json()?;

    println!("SALE: {:#?}", sale);
```

```
    // placing a bid
    let halborn_account_balance_before_bid = halborn_account.view_account().await?.balance;
    let _bid_result = halborn_account
        .call(bond_market_contract.id(), "place_a_near_bid")
        .args_json(json!({"sale_id": 0}))
        .deposit(price.0 + 1)
        .max_gas()
        .transact()
        .await?
        .into_result()?;
    let halborn_account_balance_after_bid = halborn_account.view_account().await?.balance;
    println!(
        "Balance:\n{}\n{}",
        halborn_account_balance_before_bid, halborn_account_balance_after_bid
    );

    let sale: SaleJSON = bond_market_contract
        .call("get_sale")
        .args_json(json!({"sale_id": 0}))
        .view()
        .await?
        .json()?;

    println!("\nSALE after bid: {:#?}", sale);

    // trying to merge bond 1 and 3
    halborn_account
        .call(bond_operator_contract.id(), "merge_bonds")
        .args_json(json!({
            "bond_id": 1,
            "other_id": 3,
        }))
        .transact()
        .await?
        .into_result()?;

    // waiting for over 20 seconds for the sale to end...
    println!("Waiting for the auction to end...");
    tokio::time::sleep(tokio::time::Duration::from_secs(21)).await;
```

FINDINGS & TECH DETAILS

```
    println!("Auction should be done by now...");

    // trying to complete the auction sale...
    let pull_result = halborn_account
        .call(bond_market_contract.id(), "pull_sale_bond")
        .args_json(json!({
            "sale_id": 0
        }))
        .max_gas()
        .transact()
        .await?
        .into_result()?;
    println!("\nPULL RESULT: {:#?}", pull_result);

    let sale: SaleJSON = bond_market_contract
        .call("get_sale")
        .args_json(json!({"sale_id": 0}))
        .view()
        .await?
        .json()?;

    println!("\nSale after trying to pull it: {:#?}", sale);

    let halborn_account_balance_after_trying_to_pull =
        halborn_account.view_account().await?.balance;
    println!(
        "Balance after trying to complete auction: {}",
        halborn_account_balance_after_trying_to_pull
    );

    let remove_bid_result = halborn_account
        .call(bond_market_contract.id(), "remove_loser_bid")
        .args_json(json!({"sale_id": 0}))
        .transact()
        .await?
        .into_result();
    if let Err(res) = remove_bid_result {
        println!("ERR: {}", res);
    }
    Ok(())
}
```

Recommendation:

It is recommended to implement an authorization check in the merge_bonds function so that only the user who owns both bonds can merge them. Additionally, merging and splitting bonds should be possible only for bonds that are not on sale.

Remediation Plan:

SOLVED: The MetaPool has solved this issue in commit ef7772ff by adding a verification mechanism that makes sure only the owner can merge bonds and only if neither bond is on sale.

4.2 (HAL-02) LOSS OF REWARDS DUE TO KICKSTARTER UPDATE - LOW (2.0)

Description:

Our analysis revealed a potential vulnerability in the KatherineFundraising contract, specifically in the function update_kickstarter. This function allows either the contract's owner or Kickstarter's owner to modify details pertaining to the fundraising effort. However, the implementation of the update_kickstarter function is flawed as it inadvertently resets storage variables tied to the Kickstarter, including the available_rewards_tokens.

The available_rewards_tokens variable is crucial, as it stores the quantity of project tokens (ptokens) available to be offered as rewards for supporters. The current implementation of the update_kickstarter function, however, resets this variable to zero each time it is called. As a result, it can unintentionally erase information regarding the remaining ptokens, leading to their loss.

Interestingly, this loss of reward tokens occurs even if there are no changes made to the ptoken contract itself. This issue poses a significant threat to the integrity of the fundraising efforts, as it could lead to supporters not receiving the ptokens they were promised, thereby undermining trust in the system. Further, the loss of ptokens could negatively impact the overall fundraising process.

Thus, it is essential to address this vulnerability, to ensure proper functioning of the update_kickstarter function, and maintain the accurate count of available_rewards_tokens to protect the integrity and reliability of the KatherineFundraising contract.

Code Location:

Down below is a code snippet from the internal_update_kickstarter function:

Listing 3: contracts/katherine-fundraising-contract/src/kickstarter.rs (Line 412)

```
    pub(crate) fn internal_update_kickstarter(
        &mut self,
        old_kickstarter: Kickstarter,
        name: String,
        slug: String,
        owner_id: AccountId,
        open_timestamp: EpochMillis,
        close_timestamp: EpochMillis,
        token_contract_address: AccountId,
        deposits_hard_cap: U128,
        max_tokens_to_release_per_stnear: U128,
        token_contract_decimals: u8
    ) {
        assert!(
            old_kickstarter.open_timestamp >= get_current_epoch_millis(),
            "Changes are not allow after the funding period started!"
        );

        let id = old_kickstarter.id;
        let kickstarter = Kickstarter {
            id,
            name,
            slug,
            goals: Vector::new(Keys::Goals.as_prefix(&id.to_string()).as_bytes()),
            winner_goal_id: None,
            katherine_fee: None,
            total_tokens_to_release: None,
            deposits: UnorderedMap::new(Keys::Deposits.as_prefix(&id.to_string()).as_bytes()),
            rewards_withdraw: UnorderedMap::new(
                Keys::RewardWithdraws.as_prefix(&id.to_string()).as_bytes(),
            ),
            stnear_withdraw: UnorderedMap::new(
                Keys::StnearWithdraws.as_prefix(&id.to_string()).as_bytes(),
            ),
            total_deposited: 0,
            deposits_hard_cap: deposits_hard_cap.0,
```

```
            max_tokens_to_release_per_stnear: max_tokens_to_release_per_stnear.0,
            enough_reward_tokens: false,
            owner_id,
            active: true,
            successful: None,
            stnear_price_at_freeze: None,
            stnear_price_at_unfreeze: None,
            creation_timestamp: get_current_epoch_millis(),
            open_timestamp,
            close_timestamp,
            token_contract_address,
            token_contract_decimals,
            available_reward_tokens: 0,
        };
        kickstarter.assert_timestamps();
        self.kickstarters.replace(id as u64, &kickstarter);
        self.kickstarter_id_by_slug.remove(&old_kickstarter.slug);
        self.kickstarter_id_by_slug
            .insert(&kickstarter.slug, &kickstarter.id);
    }
}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:C/R:N/S:U (2.0)

Recommendation:

It is recommended to implement an update_kickstarter function in a way that will not generate value loss in an underlying asset. If the AccoundId associated with ptoken is not changed, then the available_reward_tokens value should not be zeroed-out. On the other hand, if that AccountId changes, then returning already sent tokens to the previous owner may be considered.

Remediation Plan:

SOLVED: The MetaPool team has solved this issue in commit efadbdc7 by deprecating (and effectively deleting) the update_kickstarter function.

4.3 (HAL-03) DENIAL OF SERVICE CONDITION DUE TO STORAGE BLOATING - INFORMATIONAL (0.5)

Description:

During our analysis, we identified a potential issue with the create_vault function, which pertains to the handling of deposit amounts associated with storage fees. The function currently adds values to the contract's storage without ensuring that a sufficient deposit has been sent with the call to cover these storage costs. Importantly, the design of the create_vault function does not currently allow for a deposit to be made at the time of the call.

Without an accompanying deposit, the contract is forced to compensate for storage fees from its own free balance. If the contract's free balance is insufficient, the call to create_vault will fail due to lack of funds to cover the storage fees.

This situation presents a considerable vulnerability, as it potentially disrupts the contract's operations and the creation of new vaults. Moreover, it places an undue burden on the contract's free balance, which could have serious implications if it is not properly monitored and managed.

Code Location:

Down below is a code snippet from the create_vault function:

Listing 4: contracts/bond-operator-contract/src/lib.rs

```
pub fn create_vault(
    &mut self,
    vault_id: VaultId,
    stnear_price_at_freeze: U128,
    initial_stnear_balance: U128,
    initial_ptoken_balance: U128,
```

```
    ptoken_contract_address: String,
    stnear_freeze_timestamp: U64,
    interest_beneficiary_until_unfreeze: String,
    interest_beneficiary_near_claimed: U128,
    bond_owners_near_claimed: U128,
    bond_owners_ptoken_claimed: U128,
    stnear_vault_maturity_datetime: U64,
    ptoken_start_linear_release_datetime: U64,
    ptoken_vault_maturity_datetime: U64,
    vault_owner_id: String
) {
    self.assert_only_owner();
    self.assert_new_vault_id(&vault_id);

    let vault = Vault::new(
        vault_id.clone(),
        stnear_price_at_freeze.0,
        initial_stnear_balance.0,
        initial_ptoken_balance.0,
        ptoken_contract_address.try_into().unwrap(),
        stnear_freeze_timestamp.0,
        interest_beneficiary_until_unfreeze.try_into().unwrap(),
        interest_beneficiary_near_claimed.0,
        bond_owners_near_claimed.0,
        bond_owners_ptoken_claimed.0,
        stnear_vault_maturity_datetime.0,
        ptoken_start_linear_release_datetime.0,
        ptoken_vault_maturity_datetime.0,
        vault_owner_id.try_into().unwrap()
    );
    self.vaults.insert(&vault_id, &vault);
}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:F/S:U (0.5)

Recommendation:

To address this issue, it is recommended to revise the create_vault function to accept a deposit that can adequately cover the storage fees.

This will ensure the contract's free balance is preserved and prevent the disruption of contract operations due to insufficient funds.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commits 164fecd8 and 7e108822 by implementing a requirement for the caller to cover the storage fee associated with creating a new vault.

4.4 (HAL-04) REDUNDANT STATE VALIDATION - INFORMATIONAL (0.0)

Description:

It was observed that the KatherineFundrasing contract implements a manual assertion in 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:

Down below is a code snippet from the new function:

Listing 5: contracts/katherine-fundraising-contract/src/lib.rs (Lines 45,52)

```
45 #[init]
46 pub fn new(
47     owner_id: AccountId,
48     min_deposit_amount: U128,
49     metapool_contract_address: AccountId,
50     katherine_fee_percent: BasisPoints,
51 ) -> Self {
52     assert!(!env::state_exists(), "The contract is already initialized");
53     Self {
54         owner_id,
55         supporters: UnorderedMap::new(Keys::Supporters),
56         kickstarters: Vector::new(Keys::Kickstarters),
57         kickstarter_id_by_slug: UnorderedMap::new(Keys::KickstarterId),
58         min_deposit_amount: min_deposit_amount.0,
59         metapool_contract_address,
60         katherine_fee_percent,
61         max_goals_per_kickstarter: 5,
62         active_projects: UnorderedSet::new(Keys::Active),
63     }
64 }
```

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 redundant code.

Remediation Plan:

SOLVED: The MetaPool team has solved this issue in commit 6727d175 by removing the redundant code.

4.5 (HAL-05) REDUNDANT MANUAL CALLBACK ASSERTION - INFORMATIONAL (0.0)

Description:

The activate_successful_kickstarter_after function is marked with #[private] macro, which allows this function to only be called by the contract itself. However, it was observed that this function is also manually asserting that the predecessor_account_id is equal to current_account_id.

Code Location:

Down below is a code snippet from the assert_self function:

Listing 6: near-sdk-3.1.0/src/utils/mod.rs

```
pub fn assert_self() {
    assert_eq!(env::predecessor_account_id(), env::
↳ current_account_id(), "Method is private");
}
```

Down below is a code snippet from the activate_successful_kickstarter_after function:

Listing 7: contracts/katherine-fundraising-contract/src/internal.rs (Lines 97,103)

```
97  #[private]
98  pub fn activate_successful_kickstarter_after(
99      &mut self,
100     kickstarter_id: KickstarterId,
101     goal_id: GoalId,
102 ) {
103     assert_self();
104     assert_eq!(
105         env::promise_results_count(),
```

```
            1,
            "This is a callback method"
        );

        let st_near_price = match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Failed => panic!("Meta Pool is not available!"),
            PromiseResult::Successful(result) => {
                let price = near_sdk::serde_json::from_slice::<U128>(&result).unwrap();
                price.0
            },
        };
        let mut kickstarter = self.internal_get_kickstarter(kickstarter_id);
        match kickstarter.goals.get(goal_id as u64) {
            None => panic!("Kickstarter did not achieved any goal!"),
            Some(goal) => {
                let total_tokens_to_release = self.calculate_total_tokens_to_release(
                    &kickstarter,
                    goal.tokens_to_release_per_stnear
                );
                let katherine_fee = self.calculate_katherine_fee(total_tokens_to_release);
                assert!(
                    kickstarter.available_reward_tokens >= (total_tokens_to_release + katherine_fee),
                    "Not enough available reward tokens to back the supporters rewards!"
                );
                kickstarter.winner_goal_id = Some(goal.id);
                kickstarter.active = false;
                self.active_projects.remove(&kickstarter.id);
                kickstarter.successful = Some(true);
                kickstarter.katherine_fee = Some(katherine_fee);
                kickstarter.total_tokens_to_release = Some(total_tokens_to_release);
                kickstarter.stnear_price_at_freeze = Some(st_near_price.into());
                self.kickstarters
                    .replace(kickstarter_id as u64, &kickstarter);
            }
```

```
    }
}

```

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 redundant code.

Remediation Plan:

SOLVED: The MetaPool team has solved this issue in commit abe7e854 by removing redundant code.

4.6 (HAL-06) NOT NECESSARY MACRO USAGE - INFORMATIONAL (0.0)

Description:

Some impl blocks of KatherineFundraising contract are marked with #[near_bindgen] macro, although they define only internal functions.

Code Location:

Down below is a code snippet from the internal_create_goal function:

Listing 8: contracts/katherine-fundraising-contract/src/goal.rs (Lines 36,38,93)

```
36 #[near_bindgen]
37 impl KatherineFundraising {
38     pub(crate) fn internal_create_goal(
39         &mut self,
40         kickstarter: &mut Kickstarter,
41         name: String,
42         desired_amount: U128,
43         unfreeze_timestamp: EpochMillis,
44         tokens_to_release_per_stnear: U128,
45         cliff_timestamp: EpochMillis,
46         end_timestamp: EpochMillis,
47     ) -> GoalId {
48         kickstarter.assert_goal_status();
49         kickstarter.assert_before_funding_period();
50         kickstarter.assert_number_of_goals(self.max_goals_per_kickstarter);
51
52         let desired_amount = desired_amount.0;
53         let tokens_to_release_per_stnear = tokens_to_release_per_stnear.0;
54         let id = kickstarter.get_number_of_goals();
55         assert!(
56             kickstarter.deposits_hard_cap >= desired_amount,
57             "Desired amount must not exceed the deposits hard cap!"
```

```
        );
        assert!(
            kickstarter.max_tokens_to_release_per_stnear >=
↳ tokens_to_release_per_stnear,
            "Tokens to release must not exceed the max tokens to
↳ release per stNEAR!"
        );
        if id > 0 {
            let last_goal = kickstarter.goals.get((id - 1) as u64)
↳ .unwrap();
            assert!(
                desired_amount >= last_goal.desired_amount,
                "Next goal cannot have a lower desired amount that
↳  the last goal!"
            );
            assert!(
                unfreeze_timestamp <= last_goal.unfreeze_timestamp
↳ ,
                "Next goal cannot freeze supporter funds any
↳ longer than the last goal!"
            );
            assert!(
                tokens_to_release_per_stnear >= last_goal.
↳ tokens_to_release_per_stnear,
                "Next goal cannot release less pTOKEN than the
↳ last goal!"
            );
        }
        let goal = Goal {
            id,
            name,
            desired_amount,
            unfreeze_timestamp,
            tokens_to_release_per_stnear,
            cliff_timestamp,
            end_timestamp,
        };
        kickstarter.goals.push(&goal);
        self.kickstarters
            .replace(kickstarter.id as u64, &kickstarter);
        goal.id
    }

```

```
93     pub(crate) fn internal_delete_last_goal(&mut self, kickstarter: &mut Kickstarter) {
94         kickstarter.assert_goal_status();
95         kickstarter.assert_before_funding_period();
96         kickstarter.goals.pop();
97         self.kickstarters
98             .replace(kickstarter.id as u64, &kickstarter);
99     }
100 }
```

Down below is a code snippet from the assert_min_deposit_amount function:

Listing 9: contracts/katherine-fundraising-contract/src/deposit.rs (Lines 45,47,56,83)

```
45 #[near_bindgen]
46 impl KatherineFundraising {
47     fn assert_min_deposit_amount(&self, amount: Balance) {
48         assert!(
49             amount >= self.min_deposit_amount,
50             "minimum deposit amount is {}",
51             self.min_deposit_amount
52         );
53     }
54
55     /// Process a stNEAR deposit to Katherine Contract.
56     fn process_supporter_deposit(
57         &mut self,
58         supporter_id: &AccountId,
59         amount: &Balance,
60         kickstarter: &mut Kickstarter,
61     ) {
62         // Update Kickstarter
63         kickstarter.assert_within_funding_period();
64         kickstarter.assert_enough_reward_tokens();
65
66         let new_total_deposited = kickstarter.total_deposited + amount;
67         assert!(
68             new_total_deposited <= kickstarter.deposits_hard_cap,
69             "The deposits hard cap cannot be exceeded!"
70         );
71         kickstarter.total_deposited = new_total_deposited;
```

```
        kickstarter.update_supporter_deposits(&supporter_id, amount);
        self.kickstarters
            .replace(kickstarter.id as u64, &kickstarter);

        // Update Supporter.
        let mut supporter = self.internal_get_supporter(&supporter_id);
        supporter.supported_projects.insert(&kickstarter.id);
        self.supporters.insert(&supporter_id, &supporter);
    }

    /// Process a reward token deposit to Katherine Contract.
    fn process_kickstarter_deposit(
        &mut self,
        amount: Balance,
        kickstarter: &mut Kickstarter,
    ) {
        assert_eq!(
            &env::predecessor_account_id(),
            &kickstarter.token_contract_address,
            "Deposited tokens do not correspond to the Kickstarter contract."
        );
        assert!(
            get_current_epoch_millis() < kickstarter.close_timestamp,
            "Kickstarter Tokens should be provided before the funding period ends."
        );
        let amount = kickstarter.less_to_24_decimals(amount);
        let max_tokens_to_release = self.calculate_max_tokens_to_release(&kickstarter);
        let min_tokens_to_allow_support = max_tokens_to_release
            + self.calculate_katherine_fee(max_tokens_to_release);
        kickstarter.available_reward_tokens += amount;
        kickstarter.enough_reward_tokens = {
            kickstarter.available_reward_tokens >= min_tokens_to_allow_support
        };
        self.kickstarters
            .replace(kickstarter.id as u64, &kickstarter);
    }
}
```

```
109
```

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 unnecessary #[near_bindgen] macro usage.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 25c435f5 by removing the unnecessary macro usage.

4.7 (HAL-07) REDUNDANT FUNCTION - INFORMATIONAL (0.0)

Description:

The KatherineFundraising contract defines a delete_kickstarter function. All the function does is cause the contract to panic with information that a Kickstarter cannot be deleted. The KatherineFundraising contract does not implement any standard that would require delete_kickstarter function to be present. As a consequence, there is no value originating from this function, yet it is present in the wasm binary making it bigger, which directly impacts the deployment costs.

Code Location:

Down below is a code snippet from the delete_kickstarter function:

Listing 10: contracts/katherine-fundraising-contract/src/lib.rs (Line 363)

```
pub fn delete_kickstarter(&mut self, id: KickstarterId) {
    panic!("Kickstarter {} must not be deleted!", 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 delete unnecessary function.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 6727d175 by removing the delete_kickstarter function.

4.8 (HAL-08) DEAD CODE - INFORMATIONAL (0.0)

Description:

It was observed that the code present in the interest.rs file in the KatherineFundraising contract is completely commented out.

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 delete files that are not adding a meaningful logic implementation.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit 6727d175 by deleting the dead code.

4.9 (HAL-09) JAVASCRIPT INCOMPATIBLE TYPE - INFORMATIONAL (0.0)

Description:

It was observed that creating a kick-starter in KatherineFundraising contract requires the caller to send arguments of type u64. The contract is interacted with by JavaScript API directly or indirectly via near-cli. JavaScript does not support the whole range of u64 type, and the max value that could be represented with precision is equal to 2**53 - 1. Providing a value higher than that one will result in imprecise representation (the actual value would be different from what the user supplied) or in error. It is worth noting that values that could be impacted by this finding are associated with timestamps, and it is implausible for regular interaction to require supplying values that could break this functionality.

Code Location:

Listing 11: contracts/katherine-fundraising-contract/src/types.rs (Line 7)

```
7 pub type EpochMillis = u64;
```

Exemplary usage of EpochMillis type as user-facing function:

Listing 12: contracts/katherine-fundraising-contract/src/lib.rs (Lines 338,339)

```
333 pub fn create_kickstarter(
334     &mut self,
335     name: String,
336     slug: String,
337     owner_id: AccountId,
338     open_timestamp: EpochMillis,
339     close_timestamp: EpochMillis,
```

```
    token_contract_address: AccountId,
    deposits_hard_cap: U128,
    max_tokens_to_release_per_stnear: U128,
    token_contract_decimals: u8,
) -> KickstarterId {
    self.assert_only_owner();
    self.assert_unique_slug(&slug);
    let id = self.kickstarters.len() as KickstarterId;
    self.internal_create_kickstarter(
        id,
        name,
        slug,
        owner_id,
        open_timestamp,
        close_timestamp,
        token_contract_address,
        deposits_hard_cap,
        max_tokens_to_release_per_stnear,
        token_contract_decimals
    )
}
```

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 convert u64 into NEAR's U64 type.

Remediation Plan:

SOLVED: The MetaPool team solved this issue in commit ecaf7820 by changing the u64 type to U64 json-compatible type.

4.10 (HAL-10) 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 files of KatherineFundraising, BondOperator and BondMarket contracts specified the crate-type as both cdylib and rlib, however usually only cdylib is necessary. Specifying the crate-type to only cdylib resulted in a wasm binary size reduction of 13.1%, 11.5% and 10.8% respectively.

Code Location:

Listing 13: contracts/katherine-fundraising-contract/Cargo.toml (Line 8)

```
[lib]
crate-type = ["cdylib", "rlib"]
```

Listing 14: contracts/bond-operator-contract/Cargo.toml (Line 8)

```
[lib]
crate-type = ["cdylib", "rlib"]
```

Listing 15: contracts/bond-market-contract/Cargo.toml (Line 8)

```
[lib]
crate-type = ["cdylib", "rlib"]
```

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 delete rlib from the crate-type list.

Remediation Plan:

SOLVED: The MetaPool team has solved this issue in commit b4ba2cec by removing rlib from crate-type list.

FINDINGS & TECH DETAILS

4.11 (HAL-11) OUTDATED DEPENDENCIES - INFORMATIONAL (0.0)

Description:

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

- near-sdk
- near-contract-standards

Code Location:

Listing 16: contracts/katherine-fundraising-contract/Cargo.toml (Lines 11,12)

```
[dependencies]
near-sdk = "3.1.0"
near-contract-standards = "3.1.1"
uint = "0.9.3"
json = "0.12.4"
```

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 version.

Remediation Plan:

ACKNOWLEDGED: The MetaPool team has acknowledged this issue, and decided to keep the current dependency versions not to introduce breaking changes, since newer version of NEAR SDK introduce drastic changes in cross-contract call API.

THANK YOU FOR CHOOSING

HALBORN