PotLock

Security Assessment

February 15th, 2024 — Prepared by OtterSec

James Wang james.wang@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

PotLock engaged OtterSec to assess the `pot` and `pot-factory` programs. This assessment was conducted between February 1st and February 5th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 10 findings throughout this audit engagement.

In particular, we identified several vulnerabilities, including missed checks on donation parameters (OS-PLK-ADV-00), incorrect accounting of total matching donations (OS-PLK-ADV-01), and potential out-of-gas issues (OS-PLK-ADV-02) while processing payouts.

We also provided suggestions regarding the incorporation of additional validations and checks (OS-PLK-SUG-02) and suggested some modifications to the code base (OS-PLK-SUG-00). Furthermore, we proposed simplifying the logic to improve readability and eliminate redundancy in the code (OS-PLK-SUG-03). Throughout the engagement, the PotLock team was responsive to feedback and demonstrated a prompt approach to addressing identified issues.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/PotLock/core. This audit was performed against commit 51974b2.

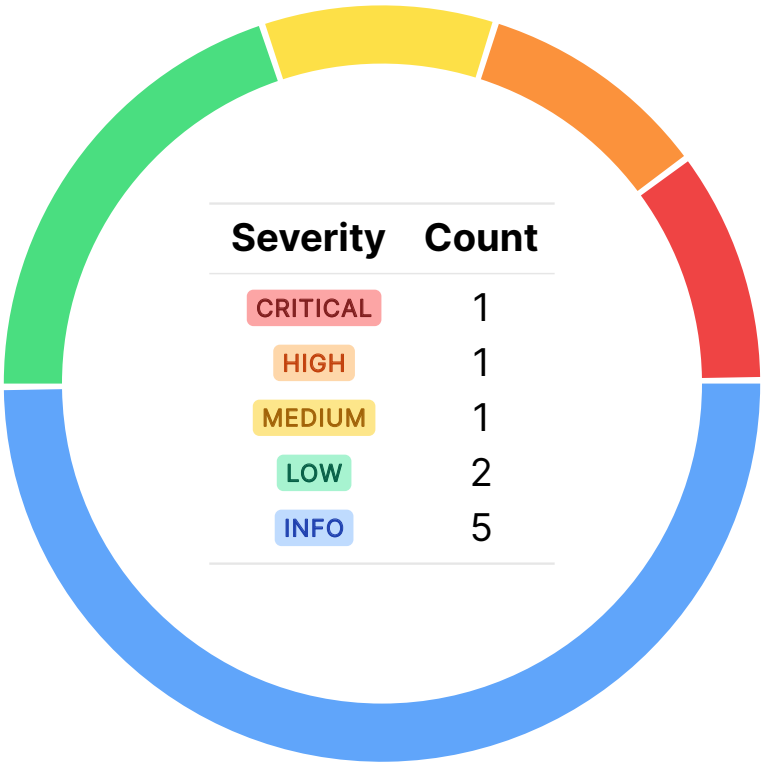
A brief description of the programs is as follows:

Name	Description
pot-factory	Contract responsible for managing the creation of new pot instances.
pot	Quadratic funding market that allows users to sign projects up for donations or donate to projects.

03 — Findings

Overall, we reported 10 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	1
MEDIUM	1
LOW	2
INFO	5

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PLK-ADV-00	CRITICAL	RESOLVED	`donate` lacks proper checks for matching donations, allowing them even when the round is inactive and failing to ensure that the `project_id` is set to `None` to prevent double payments.
OS-PLK-ADV-01	HIGH	RESOLVED	Miscalculation in `process_donation`, where the `required_deposit` is subtracted from `remainder` after updating `total_matching_pool_donations`.
OS-PLK-ADV-02	MEDIUM	RESOLVED	Possible out-of-gas issues within `payout`, during project iteration in `chef_set_payouts` and `admin_process_payouts`.
OS-PLK-ADV-03	LOW	RESOLVED	Race condition during the deployment of `pot` contracts, enabling two concurrent calls to `deploy_pot` with the same `pot_account_id_str`.
OS-PLK-ADV-04	LOW	RESOLVED	Significant centralization and governance risks are present in the current protocol implementation.

Inadequate Checks For Matching Donation CRITICAL OS-PLK-ADV-00

Description

The existing code for `donate` ensures that the round must be active for non-matching pool donations, and a `project_id` must be provided. However, this check does not apply to matching pool donations. Without this check, a potential vulnerability exists where matching pool donations might be accepted even when the round is inactive. This lack of check also allows matching pool donations to specify an associated `project_id`, counting toward both a project-specific donation and matching pool donation, resulting in double spending of the funds.

>_ *pot/src/donations.rs* rust

```rust
#[payable]
pub fn donate(
    &mut self,
    project_id: Option<ProjectId>,
    message: Option<String>,
    referrer_id: Option<AccountId>,
    matching_pool: Option<bool>,
    bypass_protocol_fee: Option<bool>,
) -> Promise {
    [...]
    if !is_matching_pool {
        self.assert_round_active();
        // error if this is an end-user donation and no project_id is provided
        if project_id.is_none() {
            env::panic_str(
                "project_id argument must be provided for public (non-matching pool) donations",
            );
        }
    }
    [...]
}
```

Remediation

Include the round activity check and ensure the `project_id` is set to `None` for matching pool donations.

Patch

Resolved in 7912082.

Incorrect Pool Balance Calculation HIGH OS-PLK-ADV-01

Description

In `process_donation` within `donations`, there is an issue in the order of operations when updating the `self.matching_pool_balance`.

>_ *pot/src/donations.rs* rust

```rust
pub(crate) fn process_donation(
    [...]
) -> DonationExternal {
    [...]
    if matching_pool {
        [...]
        self.matching_pool_balance =
            self.matching_pool_balance
                .checked_add(remainder)
                .expect(&format!(
                    "Overflow occurred when calculating self.matching_pool_balance ({} + {})",
                    self.matching_pool_balance, remainder,
                ));
    }
    [...]
    remainder = remainder.checked_sub(required_deposit).expect(&format!(
        "Overflow occurred when calculating remainder ({} - {})",
        remainder, required_deposit,
    ));
    [...]
}
```

`remainder` should be the remaining amount after deducting fees and storage costs from the total donation amount. However, subtracting the storage cost (`required_deposit`) occurs after adding it to `self.matching_pool_balance`. This would result in incorrect accounting for total matching pool donations and require the admin to fund this difference before being able to distribute payouts.

Remediation

Subtract `required_deposit` from `remainder` before updating `self.matching_pool_balance`.

Patch

Resolved in e8ef580.

Out Of Gas Error MEDIUM OS-PLK-ADV-02

Description

There is a potential out-of-gas issue in `payout` due to iterating over a large number of items, specifically the `approved_application_ids` set, in both `chef_set_payouts` and `admin_process_payouts`.

>_ *pot/src/payout.rs* rust

```rust
#[payable]
pub fn chef_set_payouts(&mut self, payouts: Vec<PayoutInput>) {
    [...]
    // clear any existing payouts (in case this is a reset, e.g. fixing an error)
    for application_id in self.approved_application_ids.iter() {
    [...]
    }
}

pub fn admin_process_payouts(&mut self) {
    [...]
    // pay out each project
    // for each approved project...
    // loop through self.approved_application_ids set
    for project_id in self.approved_application_ids.iter() {
    [...]
    }
}
```

When the number of approved projects (`approved_application_ids`) is large, iterating over the entire set in a single transaction may result in high gas consumption. The program incurs gas costs for each iteration; therefore, if the set is extensive, the cumulative gas cost may exceed the maximum gas limit allowed for a single transaction on the blockchain.

Remediation

Introduce a hard limit on the amount of applications that may be approved.

Patch

Resolved in 2b8db30.

Race Condition In Pot Deployment LOW OS-PLK-ADV-03

Description

`pots_by_id` is only populated in `deploy_pot_callback`, thus creates a potential race condition for `deploy_pot` with the same `pot_account_id` to pass existence check. The impacts of this issue are limited since only the first cross-contract call to `Pot.new` will succeed, and others will fail due to checks in the `Pot` contract. However, allowing `deploy_pot` to succeed while it should technically fail could create confusion on whether a deployment properly executes.

>_ *pot_factory/src/pot.rs* rust

```rust
pub fn deploy_pot(&mut self, mut pot_args: PotArgs) -> Promise {
    [...]

    self.pots_by_id
        .insert(&pot_account_id, &VersionedPot::Current(pot.clone())); // TODO: review this for
            ↪  race conditions

    [...]

    self.pots_by_id.remove(&pot_account_id);

    // deploy pot
    Promise::new(pot_account_id.clone())
        .create_account()
        [...]
        .function_call(
            "new".to_string(),
            serde_json::to_vec(&pot_args).unwrap(),
            0,
            XCC_GAS,
        )
        [...]
}
```

Remediation

Keep a placeholder entry in `pots_by_id` within `deploy_pot` to prevent another concurrent call with the same `pot_account_id` from succeeding.

Patch

Resolved in a920da5.

Centralization And Governance Risks LOW OS-PLK-ADV-04

Description

Several potential risks and issues were identified related to centralization and mismanagement:

1. Payout calculations occur offline, and admins maintain full discretion over the process. Compromised admin accounts could misuse this authority to steal matching donations.
2. There is no sanity check on `pot` or `pot_factory` parameters, allowing for mismanagement of round start/end times, fee percentages, and other critical parameters.
3. Configurations may be changed after the donation starts, challenging users who verify the pot contract state they are interacting with.

Remediation

1. Implement on-chain logic for payout calculations or add additional checks to ensure transparency and prevent unauthorized actions. Utilizing a multi-signature scheme or incorporating decentralized governance mechanisms may enhance security and reduce the risk of mismanagement.
2. Add validation checks for pot parameters during initialization or configuration updates. This may help prevent unintended errors or manipulations that break pots. Set critical parameters within reasonable and secure bounds.
3. Add restrictions on configuration changes once the donation round has started. This helps maintain transparency and ensures that users can trust the state of the contract during their interactions.

Patch

1. Due to the complexity of pairwise calculation, it is impractical to do calculations on-chain. Instead, a payout result challenge/resolve feature is implemented to increase transparency into payout calculation in 24e1be1, 35fe826, fe4a1c9, de2d116 and 5bfcf12.
2. Resolved in 9a867ef and 2b8db30.
3. Resolved in 79d53df by logging updates to allow frontend to display and inform users of parameter changes throughout the lifetime of `pot`.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-PLK-SUG-00	Recommendations regarding modification to the code base.
OS-PLK-SUG-01	`get_applications` lacks proper ordering of iterator operations.
OS-PLK-SUG-02	Incorporating missing validations and checks into the code base.
OS-PLK-SUG-03	Suggestions for simplifying the logic to enhance readability and efficiency while eliminating redundancy in the code base.
OS-PLK-SUG-04	`donations` has potential race conditions in handling external configurations during the donation creation process, specifically in `handle_protocol_fee` and `process_donation`

Code Refactoring

OS-PLK-SUG-00

Description

1. In `pot` and `pot_factory`, utilize `PanicOnDefault` instead of implementing `Default` since holding default values for certain fields may not make sense for the contract. `PanicOnDefault` provides a derive macro to automatically implement the `Default` trait for enumerations and structures, causing a panic when creating a default instance.
2. The protocol currently uses `start_index` and `limit` parameters when iterating over unordered data structures in `applications`, `donations` and `payouts` within `pot`. `UnorderedMap` may not provide a predictable order during iteration. Utilizing `skip` and `take` on unordered data may result in randomness, as the order of elements is not guaranteed. Instead, utilize an ordered data structure, such as a `TreeMap`.
3. Within `process_donations`, `remainder` is added to different total counters before subtracting storage fees from it. It is more appropriate to record the final `remainder` after subtracting all other fees and expenses so that it accurately reflects the remaining amount after all calculations are complete.

Remediation

Update the code base with the above-mentioned modifications.

Patch

The first and third are fixed in caddb5a and e8ef580. For the second issue, `UnorderedMap` is preferred over ordered variants for its lookup efficiency.

Improper Order Of Iterator Operations OS-PLK-SUG-01

Description

Within `get_applications` in `applications`, there is an issue concerning the order of the iterator methods applied to `application`. Specifically, the `filter` operation applies before the `take` operation.

assert_statement rust

```rust
pub fn get_applications(
    &self,
    from_index: Option<u64>,
    limit: Option<u64>,
    status: Option<ApplicationStatus>,
) -> Vec<Application> {
    [...]
    if let Some(status) = status {
        self.applications_by_id
            .iter()
            .skip(start_index as usize)
            .take(limit.try_into().unwrap())
            .filter(|(_account_id, application)| {
                Application::from(application).status == status
            })
            .map(|(_account_id, application)| Application::from(application))
            .collect()
    }
    [...]
}
```

If a user provides a specific `limit`, the take operation will restrict the number of `applications` before applying the filter based on the status. As a result, the user-provided limit may not be respected as expected. The filtering occurs on a subset of `applications` determined by `take`, potentially resulting in an incomplete or unexpected set of results.

Remediation

Apply `filter` first, to include `applications` derived from the specified criteria and then limit the results utilizing `take`.

Patch

Resolved in a48b7af.

Missing Validations

OS-PLK-SUG-02

Description

1. `get_donations_for_project`, `get_donations_for_donor` and `get_payouts`, assert that the `start_index` parameter, utilized for pagination, is within valid bounds. However, the program asserts the values against the length of incorrect structures.

```rust
>_ assert_statement
[...]
assert!(
    (self.donations_by_id.len() as u128) >= start_index,
    "Out of bounds, please use a smaller from_index."
);
[...]
let donation_ids_by_donor_set = self.donation_ids_by_donor_id.get(&donor_id).unwrap();
donation_ids_by_donor_set
    .iter()
    .skip(start_index as usize)
    [...]
```

2. In `refund_deposit`, within in `pot` and `pot_factory`, `if refund > 1` checks whether there are storage fees to refunds. `if refund > 1` should be utilized instead to ensure that `refund = 1` is not rejected by the check.

Remediation

1. Compare with the iterated data structure.
2. Modify the check in `refund_deposit`, within `pot` and `pot_factory`, from `refund > 0` to `refund > 1`.

Patch

Resolved in a55e23a, a07ec6d and 0933278.

Code Optimizations OS-PLK-SUG-03

Description

1. The `sybil_always_allow_callback`, `bypass_protocol_fee` and `assert_can_apply_callback` callbacks are unnecessary. Instead of performing an external call and waiting for a callback, invoke the internal logic of these functions directly to simplify the control flow and reduce gas consumption.
2. The state existence checks on the initialization functions: `pot::new` and `pot_factory::new` are redundant since they contain the `#[init]` attribute, which implicitly checks whether the contract state has been initialized or not.

   >_ *pot_factory/src/lib.rs* rust

   ```rust
   #[init]
   pub fn new([...]) -> Self
   {
       [...]
       assert!(!env::state_exists(), "Already initialized");
       [...]
   }
   ```

3. Within `admin` in `pot` and `pot_factory`, several instances have been identified where the program performs unnecessary storage fee checks/refunds on actions that would not affect storage utilization.
4. The program does not utilize `default_chef_fee_basis_points` parameter within `pot_factory` on `pot` deploy.

Remediation

Remove all unnecessary or redundant checks and functionalities mentioned above.

Patch

Resolved in 07a6f8c, 829bd7b, 98c09fb, 692d0f6, 7e216de and 9387821.

Presence Of Race Condition During Donation OS-PLK-SUG-04

Description

`handle_protocol_fee` queries `protocol_config_provider` to obtain the basis points and account ID for the protocol fee. This asynchronous operation may introduce a race condition if there is a change in the configuration provider (`protocol_config_provider`) between the time of the call to `donate` and the actual call to the provide. Similarly, `chef_fee` is fetched from configuration and applied in `process_donation`. These may allow for unforeseen expenses for the `doner` in the form of increased fee rates.

>_ *pot/src/donations.rs* rust

```
    pub(crate) fn handle_protocol_fee(
        [...]
    ) -> Promise {
        if bypass_protocol_fee.unwrap_or(false) {
         [...]
        } else if let Some(protocol_config_provider) = self.protocol_config_provider.get() {
            let (contract_id, method_name) = protocol_config_provider.decompose();
            let args = json!({}).to_string().into_bytes();
            Promise::new(AccountId::new_unchecked(contract_id.clone()))
                .function_call(method_name.clone(), args, 0, XCC_GAS)
                .then(
                    Self::ext(env::current_account_id())
                        .with_static_gas(XCC_GAS)
                        .handle_protocol_fee_callback(
                            deposit,
                            project_id,
                            message,
                            referrer_id,
                            matching_pool,
                        ),
                )
        }
        [...]
    }
```

Remediation

Fetch the configuration values in `donation` and use those pre-fetched values in later stages.

Patch

Upper bounds are enforced on fee configuration in 2b8db30, which limits the impact configuration changes may have on users.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.