DITTO

Ditto Finance

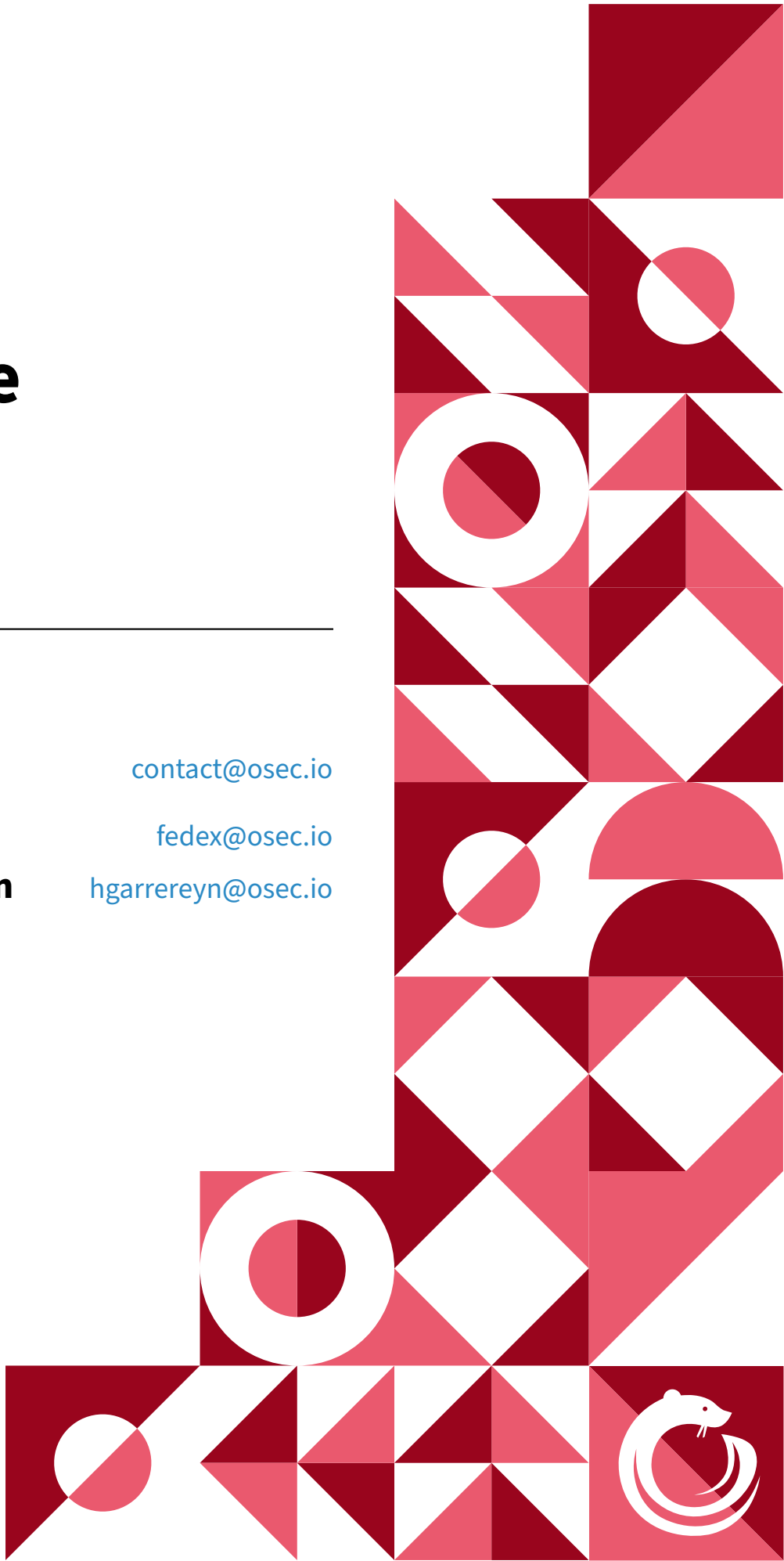
Audit

Presented by:

OtterSec contact@osec.io

Fineas Silaghi fedex@osec.io

Harrison Green hgarrereyn@osec.io



Contents

Appendices

01 | Executive Summary

Overview

Ditto Finance engaged OtterSec to perform an assessment of the `ditto-staking` program. This assessment was conducted between September 7th and September 30th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches October 14th, 2022.

Key Findings

During this engagement, we identified 6 findings.

Since the protocol is still in development, we focused on areas fundamental to the design and of high-security relevance. In particular, we identified (OS-DIT-ADV-00) which would allow a malicious attacker to saturate the computation limit on instructions and potentially freeze the protocol.

We also made a number of suggestions around code redundancy and improved error messages. Overall, the Ditto Finance team was responsive and a pleasure to work with.

02 | Scope

The source code was delivered to us in a git repository at github.com/dittosis/ditto-research-ditto-staking. This audit was performed against commit `6e1a22e` .

There was a total of one program included in this audit. A brief description of the programs is as follows.

Name	Description
ditto-staking	Liquid staking protocol that allows users to delegate APT to validators, in exchange for `stAPT`.

03 | Findings

Overall, we report 6 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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.

Severity	Count
Critical	0
High	0
Medium	1
Low	1
Informational	4

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-DIT-ADV-00	Medium	Resolved	Unbounded validator sets may exceed computation limits and halt the protocol.
OS-DIT-ADV-01	Low	Resolved	Instant unstake fee rounds to zero for small withdraws.

OS-DIT-ADV-00 [med] [resolved] | Unrestricted Validator Registration May Lead To DoS

Description

Validators can join the Ditto protocol by invoking `ditto_staking::add_validator`. When the validator whitelist is disabled, there are no restrictions on validator entry. Each new validator occupies space in the `ValidatorState` table and `ValidatorLockupBuffer` stored on the `@ditto_staking` account. Many of the computations that interact with validator state run linear time algorithms over these structures and therefore require gas usage roughly linear to the number of validators.

A malicious user could register a large number of fake validators in order to increase the usage of the associated validator tables and therefore increase the computation requirement on all subsequent instructions.

In the worst case, a malicious user may be able to register enough validators to hit the computation limit and therefore prevent the protocol from operating entirely.

Remediation

Impose a hard limit on the number of validators. Additionally, would recommend requiring all active validators to maintain a minimum level of activity or stake such that a malicious user cannot easily add a bunch of fake validators.

Patch

After discussion with the team, Ditto will be launching with their whitelist feature enabled for the foreseeable future to mitigate against such attack vectors. Ditto also implemented a `max_n_validators` configuration parameter which sets a hard limit on the number of validators.

OS-DIT-ADV-01 [low] [resolved] | Instant Unstake Fee Can Be Bypassed

Description

Users can unstake their `stAPT` instantly or delayed. In order to unstake instantly, a fee, which is defined in the configuration parameter `instant_unstake_fee_bps` , is charged. The fee is computed using the `utils::calculate_fee` function:

sources/utils.move

```
public fun calculate_fee(amount: u64, fee_bps: u64): u64 {
    amount * fee_bps / BPS_DENOMINATOR
}
```

This allows an attacker to pay no fees for small amounts, because the fee rounds down to zero. More specifically, whenever the `amount < (BPS_DENOMINATOR / fee_bps)`, the fee can be avoided.

Proof of Concept

Consider `instant_unstake_fee_bps = 30` basis points (0.3%). An attacker can transfer up to 333 atomic units of aptos, instantly, without being charged.

Remediation

Round fee computation up to ensure it is not profitable for an attacker to bypass the fee in this manner.

Patch

Fixed in bea16d3.

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 do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-DIT-SUG-00	Accessor functions in ditto_config could be improved to reduce redundancy.
OS-DIT-SUG-01	Use explicit assertions to throw specific errors on instruction failure.
OS-DIT-SUG-02	Add a mechanism to remove validators from the Ditto protocol.
OS-DIT-SUG-03	The protocol does not efficiently distribute stake between validators.

OS-DIT-SUG-00 [resolved] | Code Redundancy In ditto_config

Description

The `ditto_config` module manages the `DittoConfig` struct and provides getters and setters for the various properties. However, the object's accessors impose a verbose API:

sources/ditto_config.move

```
public fun get_pool_buffer_pct(config: &DittoConfig): u64 {
    config.pool_buffer_pct
}

ditto_config::get_pool_buffer_pct(&ditto_config::get())
```

Remediation

Consider consolidating these resource accessors such that relevant properties can be accessed with one function call:

sources/ditto_config.move

```
public fun get_pool_buffer_pct(): u64 {
    get().pool_buffer_pct
}

ditto_config::get_pool_buffer_pct()
```

Patch

Fixed in dd600b6.

OS-DIT-SUG-01 [resolved] | Add Explicit Assertions to Improve Error Messages

Description

Several functions in `ditto_staking` already contain explicit assertions to check that the protocol has been initialized:

```rust
assert!(is_ditto_initialized(), ERR_DITTO_POOL_NOT_INITIALIZED);
```

Consider adding this assertion to the following functions in `ditto_staking` which also require the protocol to be initialized as a prerequisite:

- `whitelist_validator`
- `update_ditto_state`
- `distribute_unstaked_coins`
- `join_validator_set`

Patch

Fixed in 2adc49f.

OS-DIT-SUG-02 | Add Mechanism to Remove Validators

Description

Once a validator joins the pool, its `OwnerCapability` is stored by the Ditto protocol and it is not possible for the validator to leave the protocol or recover its `OwnerCapability`.

Remediation

Consider adding a mechanism for validators to remove themselves (and/or be kicked out) of the Ditto protocol.

OS-DIT-SUG-03 | Inefficient Allocation of Resources

Description

The protocol does not consider performance of validators when assigning stake. Frozen or broken validators will receive stake despite not contributing to generating rewards.

Additionally, there is no inbuilt mechanism to rebalance stake between old validators and newly joined validators. Natural deposits and withdrawals will have the effect of rebalancing stake but at a much slower rate.

Remediation

Consider adding a mechanism to track validator performance and distribute stake proportionally. Consider adding a mechanism to rebalance stake between existing validators.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation