Bucket Protocol

Audit

Presented by:

OtterSec contact@osec.io

Akash Guruguntl Sud0u53r.ak@osec.io

Robert Chen r@osec.io

Contents

01 Executive Summary **2**
Overview . . . 2
Key Findings . . . 2

02 Scope **3**

03 Findings **4**

04 Vulnerabilities **5**
OS-BKT-ADV-00 [crit] | Improper Conversion . . . 6
OS-BKT-ADV-01 [high] | Users Unable To Claim Surplus . . . 7
OS-BKT-ADV-02 [med] | Improper Tank Value Update . . . 8
OS-BKT-ADV-03 [med] | Improper Stake Update . . . 9
OS-BKT-ADV-04 [med] | Precision Loss In Redistribution . . . 10
OS-BKT-ADV-05 [low] | Improper Token Weight Calculation . . . 11

05 General Findings **12**
OS-BKT-SUG-00 | Unnecessary Extra Reference . . . 13
OS-BKT-SUG-01 | Round Up Fee Amount Calculations . . . 14
OS-BKT-SUG-02 | Use Of Hard-Coded Values . . . 15
OS-BKT-SUG-03 | Handle Zero Debt Case For TCR . . . 16
OS-BKT-SUG-04 | Avoid Precision Loss . . . 17

Appendices

A Vulnerability Rating Scale **18**

B Procedure **19**

01 | Executive Summary

Overview

Bucket Protocol engaged OtterSec to perform an assessment of the `v1-core` program. This assessment was conducted between June 2nd and June 14th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 11 findings in total.

In particular, we have found issues related to improper amount conversions (OS-BKT-ADV-00), improper updation of values (OS-BKT-ADV-02, OS-BKT-ADV-03), and precision loss issues (OS-BKT-ADV-04).

We also made recommendations around unnecessary reference borrowings (OS-BKT-SUG-00), avoiding anti-patterns in the code (OS-BKT-SUG-02), and unnecessary precision losses (OS-BKT-SUG-04).

02 | Scope

The source code was delivered to us in a git repository at github.com/Bucket-Protocol/v1-core. This audit was performed against commit 0ad3cb5.

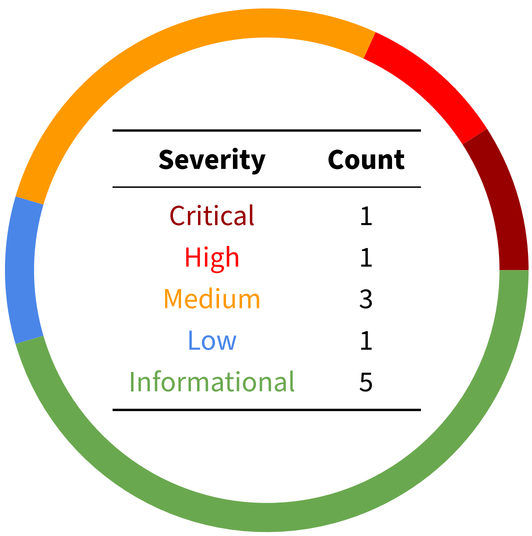
A brief description of the programs is as follows.

Name	Description
v1-core	CDP protocol built on Sui network where users may: 1. Deposit $SUI as collateral and borrow $BUCK. 2. Repay with $BUCK and take back collateral in the form of $SUI. 3. Redeem 1:1 value of $SUI from protocol using $BUCK. 4. Deposit $BUCK to tank to earn incentive token $BKT. 5. Provide liquidity for SUI/BUCK on DEX and also earn $BKT. 6. Stake $BKT to share protocol revenue, which comes from borrow fees, redemption fees, and flash-loan fees.

03 | Findings

Overall, we reported 11 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 help mitigate future vulnerabilities.



Severity	Count
Critical	1
High	1
Medium	3
Low	1
Informational	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-BKT-ADV-00	Critical	Resolved	Conversion from debt amount to collateral amount is improper.
OS-BKT-ADV-01	High	Resolved	Users are unable to claim surplus collateral from Bottles liquidated in recovery mode.
OS-BKT-ADV-02	Medium	Resolved	Improper updation of `start_s` and `start_g` values in `ContributorToken` leads to inconsistency.
OS-BKT-ADV-03	Medium	Resolved	The stake of the last Bottle is not updated, leading to inconsistency in stake amounts.
OS-BKT-ADV-04	Medium	Resolved	Precision loss in the redistribution of collateral and debt amounts to users.
OS-BKT-ADV-05	Low	Resolved	Token weight is improperly calculated by unnecessarily reducing the user's compounded stake.

OS-BKT-ADV-00 [crit] | Improper Conversion

Description

`record_repay_capped` in the `bottle` module calculates the collateral amount returned for a given debt amount.

protocol/sources/bottle.move RUST

```rust
public(friend) fun record_repay_capped<T>(bottle: &mut Bottle, repay_amount: u64,
    ↪  oracle: &BucketOracle, clock: &Clock): (bool, u64) {
    if (repay_amount >= bottle.buck_amount) {
        let (price, denominator) = bucket_oracle::get_price<T>(oracle, clock);
        // collateral: at most 110% debt
        let return_sui_amount = mul_factor(repay_amount * 110 / 100, denominator,
    ↪   price);
        bottle.collateral_amount = bottle.collateral_amount - return_sui_amount;
        bottle.buck_amount = 0;
        // fully repaid
        (true, return_sui_amount)
    } else {
        let return_sui_amount = mul_factor(bottle.collateral_amount, repay_amount,
    ↪   bottle.buck_amount);
        bottle.collateral_amount = bottle.collateral_amount - return_sui_amount;
        bottle.buck_amount = bottle.buck_amount - repay_amount;
        // not fully repaid
        (false, return_sui_amount)
    }
}
```

If the debt amount (`repay_amount`) is greater than or equal to the Bottle debt, the collateral returned is calculated as 1.1 times the debt amount. However, while converting the debt amount to the collateral amount, the amount is not adjusted based on the decimals of the collateral token, leading to an improper value of the collateral amount (`return_sui_amount`).

Remediation

Correctly convert the amount based on the decimals of the collateral token.

Patch

Fixed in 2b68221 by correctly calculating `return_sui_amount`.

OS-BKT-ADV-01 [high] | Users Unable To Claim Surplus

Description

`record_repay_capped` in the `bottle` module calculates the collateral amount to return for a given debt amount.

protocol/sources/bottle.move RUST

```
if (repay_amount >= bottle.buck_amount) {
    let (price, denominator) = bucket_oracle::get_price<T>(oracle, clock);
    // collateral: at most 110% debt
    let return_sui_amount = mul_factor(repay_amount * 110 / 100, denominator,
    ↪  price);
    bottle.collateral_amount = bottle.collateral_amount - return_sui_amount;
    bottle.buck_amount = 0;
    // fully repaid
    (true, return_sui_amount)
} else {
```

protocol/sources/bucket.move RUST

```
let bottle = bottle::borrow_bottle_mut(&mut bucket.bottle_table, debtor);
let (is_fully_repaid, return_amount) = bottle::record_repay_capped<T>(bottle,
    ↪  buck_input_amount, oracle, clock);
bottle::update_stake_and_total_stake_by_debtor(&mut bucket.bottle_table, debtor);
if (is_fully_repaid) {
    bottle::destroy_bottle(&mut bucket.bottle_table, debtor);
};
```

When the debt amount (`repay_amount`) is greater than or equal to the Bottle debt, after calculating the collateral amount to return, the `bottle.collateral_amount` subtracts from it and returns `true`. That signifies the clearing of all debt. Now, the Bottle is destroyable. However, simply destroying the Bottle deletes it from the Bottle table, which results in the user being unable to claim their surplus collateral amount from the Bottle.

Remediation

Store the surplus amount in another field before destroying the Bottle to provide a way for the user to claim their surplus collateral.

Patch

Fixed in 7b27bbf by adding another field to track the surplus amounts of users and providing a function for users to collect their surplus amounts.

OS-BKT-ADV-02 [med] | Improper Tank Value Update

Description

`claim_collateral` in the `tank` module claims the collateral gained from the liquidations. After claiming the collateral on a `ContributorToken`, `start_s` updates to indicate the claim of collateral up to that point. However, while updating the value of `start_s`, its value is set to one less than the value used for calculating the `collateral_amount` (excluding `sec_portion`).

protocol/sources/tank.move RUST

```
let sec_protion = *next_s_cache / constants::scale_factor();
let collateral_amount = mul_factor(
    token.deposit_amount,
    *s_cache - token.start_s + sec_protion,
    token.start_p,
);
token.start_s = *s_cache;
```

Similarly, `claim_bkt` claims the Bucket rewards provided by the protocol to the Tank. After claiming $BKT rewards on a `ContributorToken`, the `start_g` value becomes a value less than the value used for calculating the `bkt_output_amount` (excluding `sec_portion`).

protocol/sources/tank.move RUST

```
let sec_protion = *next_g_cache / constants::scale_factor();
let bkt_output_amount = mul_factor(
    token.deposit_amount,
    *g_cache - token.start_g + sec_protion,
    token.start_p,
);
token.start_g = *g_cache;
```

Remediation

Set the `start_s` and `start_g` to values used during the amount calculations (that includes the `sec_portion`).

Patch

Fixed in dd49e5e.

OS-BKT-ADV-03 [med] | Improper Stake Update

Description

`handle_redeem` in the `bucket` module handles the redemption of $BUCK by taking collateral from the bottles in ascending order of their collateral ratio.

protocol/sources/bucket.move RUST

```rust
} else {
    let redeemed_amount =
     ↪  compute_buck_value_to_collateral(remaining_redemption_amount,
     ↪  bucket.collateral_decimal, price, denominator);
    bottle::record_redeem(&mut bottle, redeemed_amount,
     ↪  remaining_redemption_amount);
    balance::join(&mut collateral_output, balance::split(&mut
     ↪  bucket.collateral_vault, redeemed_amount));
    bottle::insert(&mut bucket.bottle_table, debtor, bottle, insertion_place);
    remaining_redemption_amount = 0;
    break
};
// update the debtor's stakes
bottle::update_stake_and_total_stake_by_debtor(&mut bucket.bottle_table, debtor);
```

When redeeming Bottles, the `else` case inside the `while` loop handles the last Bottle's redemption. When the remaining redemption amount is less than the Bottle's buck amount, the loop ends in the `else` case with a `break` and skips the call to `bottle::update_stake_and_total_stake_by_debtor` on the last Bottle.

Remediation

Call `bottle::update_stake_and_total_stake_by_debtor` before the `break` statement in the `else` case.

Patch

Fixed in 2b68221.

OS-BKT-ADV-04 [med] | Precision Loss In Redistribution

Description

`record_redistribution` in the `bottle` module handles the redistribution of collateral and debt amounts to all Bottle users; this is done by dividing the collateral and debt amounts with the total stake amount and adding it to the accumulators.

protocol/sources/bottle.move RUST

```rust
public(friend) fun record_redistribution(
    table: &mut BottleTable,
    collateral_amount: u64,
    debt_amount: u64,
) {
    table.reward_per_unit_stake = table.reward_per_unit_stake + collateral_amount /
    ↪  table.total_stake;
    table.debt_per_unit_stake = table.debt_per_unit_stake + debt_amount /
    ↪  table.total_stake;
}
```

Since the accumulators are not factored by some value, directly dividing the collateral and debt amounts with total stake leads to less precise rounded-down values, which the accumulators add and lead to imprecise accumulation.

Remediation

Factor the collateral and debt accumulators with some value to avoid precision loss.

Patch

Fixed in b2daf7f.

OS-BKT-ADV-05 [low] | Improper Token Weight Calculation

Description

`get_token_weight` in the `tank` module calculates the weight of the user's deposit. Calculating the amount able to be withdrawn by the user uses this token weight.

protocol/sources/tank.move RUST

```
    // TODO: check this line is necessary
    if (compound_stake < token.deposit_amount/ constants::scale_factor()) {
        return 0
    };

    (compound_stake)
}
```

In this function, if the total calculated `compound_stake` of the user for the two scales is less than `token.deposit_amount/constants::scale_factor()` value, zero is returned. This results in unnecessarily reducing the user's compounded stake.

Remediation

Remove the `if` case that returns zero if the `compound_stake` is less than `token.deposit_amount/constants::scale_factor()`.

Patch

Fixed in 3a995b0.

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 lead to security issues in the future.

ID	Description
OS-BKT-SUG-00	Unnecessary extra reference for Bottle.
OS-BKT-SUG-01	Fee amounts should round up to avoid loss for protocol.
OS-BKT-SUG-02	Use of hard-coded values in the code base instead of obtaining them.
OS-BKT-SUG-03	Currently does not handle the case where total debt is zero while calculating TCR.
OS-BKT-SUG-04	Precision loss when calculating the remaining collateral amount.

OS-BKT-SUG-00 | Unnecessary Extra Reference

Description

`update_stake_and_total_stake_by_debtor` in the `bottle` module borrows the reference for Bottle twice, once each for getting and setting the stake amount. Avoid taking the reference twice by taking the mutable reference once and using it to get and set the value of the stake amount for the bottle.

Remediation

Take the mutable reference once and use it to get and set the value of the stake amount for the bottle.

OS-BKT-SUG-01 | Round Up Fee Amount Calculations

Description

When calculating the fee amounts in multiple places in the code base, `mul_factor` is used. This function rounds down the value by default. To avoid small losses for the protocol, round the values up when calculating the fee amounts.

Remediation

Round up the values while calculating the fee amounts.

OS-BKT-SUG-02 | Use Of Hard-Coded Values

Description

The code base uses hard-coded values like `110` (for MCR). In the future, if the constant value changes, it would require the developer to change all the instances of the hard-coded values.

Remediation

Obtain values (such as MCR) programmatically and use that instead.

OS-BKT-SUG-03 | Handle Zero Debt Case For TCR

Description

`get_bucket_tcr` in the `bucket` module gets the total collateral ratio of the Bucket. This function does not handle the case where the total minted $BUCK amount (debt amount) is zero and raises an error.

Remediation

Handle the case where the debt amount is zero by returning `constants::max_u64()`.

OS-BKT-SUG-04 | Avoid Precision Loss

Description

`handle_redistribution` in the `bucket` module handles the collateral and debt redistribution to the users. The calculation of the remaining collateral after taking out the fee and rebate amount is improper, giving less precise values.

Remediation

Calculate the `collateral_amount` as `collateral_amount - (2 * rebate_amount)` to avoid precision loss.

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

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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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.