code4rena.com

Opus

88–111 minutes

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Opus smart contract system written in Cairo. The audit took place between January 9 — February 6, 2024.

Wardens

28 Wardens contributed reports to Opus:

1. bin2chen
2. minhquanym
3. 0xTheC0der
4. etherhood
5. ABAIKUNANBAEV

6. zigtur
7. kfx
8. jasonxiale
9. 3docSec
10. TrungOre
11. nmirchev8
12. Aymen0909
13. kodyvim
14. aariiif
15. mahdikarimi
16. fouzantanveer
17. ZanyBonzy
18. invitedtea
19. hals
20. lsaudit
21. Kaysoft
22. 0xepley
23. Sathish9098
24. hassanshakeel13
25. yongskiws
26. LinKenji
27. hunter_w3b
28. catellatech

This audit was judged by 0xsomeone.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 9 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Opus repository, and is composed of 15 smart contracts written in the Cairo programming language and includes 4056 lines of Cairo code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced

throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (4)

[H-01] Neglect of exceptional redistribution amounts in `withdraw_helper` function

Submitted by Aymen0909, also found by etherhood, bin2chen, jasonxiale, minhquanym, and kodyvim

Lines of code

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/shrine.cairo#L1382-L1392
https://github.com/code-423n4/2024-01-opus/blob/main/src/core/shrine.cairo#L1421-L1431

Description

The `withdraw_helper` function in the `shrine` contract handles withdrawal logic for both the `withdraw` and `seize` functions. It is responsible for updating trove balances, total yang balances, and charging interest for the trove via the `charge` function. However, there is an oversight in the current implementation:

```
fn withdraw_helper(ref self: ContractState, yang:
ContractAddress, trove_id: u64, amount: Wad) {
    ...

    let new_trove_balance: Wad = trove_balance -
amount;
    let new_total: Wad =
```

```
self.yang_total.read(yang_id) - amount;

    self.charge(trove_id);

    //@audit will not account for exceptional
redistribution added to deposits balance in
`charge` call

    self.yang_total.write(yang_id, new_total);
    self.deposits.write((yang_id, trove_id),
new_trove_balance);

    // Emit events
    self.emit(YangTotalUpdated { yang, total:
new_total });
    self.emit(DepositUpdated { yang, trove_id,
amount: new_trove_balance });
}
```

The issue in the code above is that the `withdraw_helper` function proceeds to update the storage variables `yang_total` and `deposits` using the previously calculated `new_total` and `new_trove_balance` values, without accounting for any new yang balance added to the trove after an exceptional redistribution. This results in neglecting any exceptional redistributions added to the `deposits` balance during the `charge` call :

```
fn charge(ref self: ContractState, trove_id: u64)
{
    ...

    // If there was any exceptional
```

```
redistribution, write updated yang amounts to
trove
    if updated_trove_yang_balances.is_some() {
        let mut updated_trove_yang_balances =
updated_trove_yang_balances.unwrap();
        loop {
            match
updated_trove_yang_balances.pop_front() {
                Option::Some(yang_balance) => {
                    //@audit will updated the
trove yang balance

self.deposits.write((*yang_balance.yang_id,
trove_id), *yang_balance.amount);
                },
                Option::None => { break; },
            };
        };
    }


    ...
}
```

Because the trove `deposits` map is changed in the `charge` function but `withdraw_helper` uses directly the value `new_trove_balance`, which was calculated before the `charge` call, the exceptional redistribution added to `deposits` will be overridden and will be neglected in the trove yang balance.

This oversight could result in financial losses for all protocol users. When users withdraw yang amounts, any exceptional redistributions that should have been added to their trove balances

will be neglected and lost.

Impact

Users are at risk of losing all yang exceptional redistribution amounts due to an error in the `withdraw_helper` function, which causes it to neglect any yang-added redistribution to the trove `deposits` map.

Proof of concept

Let's take a simple scenario to highlight this issue:

- Bob wants to withdraw a 100 amount of yang (`yang_id`) from his trove (`trove_id`) certain, we had the following state before the tx:

  `deposits(yang_id, trove_id) = 1000`

- When Bob calls `abbot.withdraw`, `withdraw_helper` will be invoked under the hood in the shrine contract which will first calculate the new yang trove balance:

  `new_trove_balance = trove_balance - amount = 1000 - 100 = 900`

- An exceptional redistribution did happen so when the `charge` function is called it will update the trove yang `deposits` balance, so now we have (suppose redistribution is 50 yang per trove, for example):

  `deposits(yang_id, trove_id) = 1000 + 50 = 1050`

- After calling `charge`, the `withdraw_helper` function will set the trove yang balance, `v`, to the previously calculated `new_trove_balance`, so we will have:

  `deposits(yang_id, trove_id) = 1000 - 100 = 900`

- We see that the yang amount added from exceptional redistribution is completely neglected as we should have:

  `deposits(yang_id, trove_id) = 1000 - 100 + 50 = 950`

  Thus, as demonstrated in this simplified example, the issue will cause the loss of any exceptional redistribution amounts for the users resulting in a financial losses.

Tools Used

VS Code

Recommended Mitigation

To address this issue, the `charge` function should be called before calculating the new trove yang balance (`new_trove_balance`). This ensures that any exceptional redistributions are accounted for before updating the trove balance and total yang balance:

```
fn withdraw_helper(ref self: ContractState, yang:
ContractAddress, trove_id: u64, amount: Wad) {
        let yang_id: u32 =
self.get_valid_yang_id(yang);

        //@audit add exceptional redistribution
before calculating `new_trove_balance`
++      self.charge(trove_id);

        // Fails if amount > amount of yang
deposited in the given trove
        let trove_balance: Wad =
self.deposits.read((yang_id, trove_id));
```

```
        assert(trove_balance >= amount, 'SH:
Insufficient yang balance');

        let new_trove_balance: Wad =
trove_balance - amount;
        let new_total: Wad =
self.yang_total.read(yang_id) - amount;

--      self.charge(trove_id);

        self.yang_total.write(yang_id,
new_total);
        self.deposits.write((yang_id, trove_id),
new_trove_balance);

        // Emit events
        self.emit(YangTotalUpdated { yang, total:
new_total });
        self.emit(DepositUpdated { yang,
trove_id, amount: new_trove_balance });
    }
```

Assessed type

Context

tserg (Opus) confirmed and commented via duplicate issue #211:

> This is valid - potentially fixed.

0xsomeone (judge) commented:

> The warden has demonstrated how an exception trove redistribution will not be properly tracked by the withdrawal helper, resulting in an unsynchronized accounting state for the Opus

> system whereby the user will lose the collateral they acquired in the redistribution.
>
> I believe a high-risk severity is appropriate as it details a scenario in which the collateral balances of users will potentially lose the full redistributed collateral.

[H-02] `convert_to_yang_helper()` loss precision

Submitted by bin2chen

In `gate.cairo`, when the user calls `deposit()`, it calculates the corresponding shares through `convert_to_yang_helper()`. The code is as follows:

```
        fn convert_to_yang_helper(self:
@ContractState, asset_amt: u128) -> Wad {
            let asset: IERC20Dispatcher =
self.asset.read();
            let total_yang: Wad =
self.get_total_yang_helper(asset.contract_address)

            if total_yang.is_zero() {
                let decimals: u8 =
asset.decimals();
                // Otherwise, scale `asset_amt`
up by the difference to match `Wad`
                // precision of yang. If asset is
of `Wad` precision, then the same
                // value is returned
                fixed_point_to_wad(asset_amt,
decimals)
            } else {
```

```
@>              (asset_amt.into() * total_yang) /
get_total_assets_helper(asset).into()
            }
        }
```

The calculation formula is: `(asset_amt.into() * total_yang) / get_total_assets_helper(asset).into()`.

The actual calculation of converting Wad to pure numbers is: `(asset_amt * total_yang / 1e18) * 1e18 / total_assets`.

The above formula `(asset_amt * total_yang / 1e18)` will lose precision, especially when the asset's decimals are less than 18.

Assume btc as an example, decimals = 8 after `add_yang(btc)` `INITIAL_DEPOSIT_AMT = 1000` so:

`total_assets = 1000`
`total_yang = 1000e10 = 1e13`

If the user deposits 0.0009e8 BTC, according to the formula = `(asset_amt * total_yang / 1e18)`:

`= 0.0009e8 * 1e13 /1e18 = 0.9e5 * 1e13 /1e18 = 0`

With BTC's price at 40,000 USD, 0.0009e8 = 36 USD. The user will lose 36 USD.

We should cancel dividing by 1e18 and then multiplying by 1e18, and calculate directly: shares = `asset_amt.into() * total_yang.into() / total_assets.into()`.

shares = `0.0009e8 * 1e13 / 1000 = 0.0009e18 = 900000000000000`

Note: In order to successfully deposit should be > 0.0009e8 such as 0.0019e8, which is simplified and convenient to explain.

Impact

Due to the premature division by 1e18, precision is lost, and the user loses a portion of their funds.

Proof of Concept

Add to `test_abbot.cairo`:

```
    #[test]
    fn test_wad() {
        let INITIAL_DEPOSIT_AMT: u128 = 1000;
        let decimals:u8 = 8;
        let asset_amt:u128 = 90_000;
        let total_yang:Wad =
fixed_point_to_wad(INITIAL_DEPOSIT_AMT,
decimals);
        let total_assets:Wad =
INITIAL_DEPOSIT_AMT.into();
        let result:Wad = asset_amt.into() *
total_yang / total_assets;
        assert(result.into() == 0,' no zero');
        let result2_u:u256 = asset_amt.into() *
total_yang.into() / total_assets.into();
        let result2:Wad = Wad {
val:result2_u.try_into().expect('u128')};
        assert(result2.into() ==
900000000000000,' result2 no zero');
    }
```

```
$ scarb test -vvv test_wad
```

```
Running 1 test(s) from src/
[PASS]
opus::tests::abbot::test_abbot::test_abbot::test_w
(gas: ~17)
Tests: 1 passed, 0 failed, 0 skipped, 0 ignored,
390 filtered out
```

Recommended Mitigation

```
        fn convert_to_yang_helper(self:
@ContractState, asset_amt: u128) -> Wad {
            let asset: IERC20Dispatcher =
self.asset.read();
            let total_yang: Wad =
self.get_total_yang_helper(asset.contract_address)

            if total_yang.is_zero() {
                let decimals: u8 =
asset.decimals();
                // Otherwise, scale `asset_amt`
up by the difference to match `Wad`
                // precision of yang. If asset is
of `Wad` precision, then the same
                // value is returned
                fixed_point_to_wad(asset_amt,
decimals)
            } else {
-               (asset_amt.into() * total_yang) /
get_total_assets_helper(asset).into()
+               let result:u256 =
asset_amt.into() * total_yang.into() /
```

```
total_assets.into();
+                Wad {
val:result.try_into().expect('u128')};
            }
        }
```

Assessed type

Decimal

tserg (Opus) confirmed

0xsomeone (judge) commented:

> The warden has demonstrated how the “hidden” operations of multiplication and division that are performed as part of the overloaded `Wad` data type primitive operators can result in loss of precision for assets with less than 18 decimals; which are explicitly meant to be supported by the Opus system per the onboarding guidelines.
>
> I consider a high-risk rating appropriate given that the truncation will be greater the lower the decimals of the token and the higher the value per unit of the token is.

[H-03] A user can steal from the shrine by forcing redistribution of their trove; due to incorrect logic trove debt will be reset but yangs kept

Submitted by kfx, also found by bin2chen, minhquanym, and TrungOre

Let’s assume two yangs in the system, yang A and yang B, and two users:

- User U1 with trove #1 with zero A units, 1000 B units, and 500 yin

debt;

- User U2 with trove #2 10000 A unit, 1000 B units, and 500 yin debt.

  If the user U1 can force redistribution of their position, then they can steal from the shrine due to a bug in the code. The function `redistribute_helper` loops through all yangs in order, including those not in the trove #1. Since `trove_yang_amt.is_zero()` returns `true` for yang A, the `updated_trove_yang_balances` array is updated early and then `continue` statement is executed.

  However, since the `new_yang_totals` array is not updated in the iteration of the loop, some values of `updated_trove_yang_balances` end up never being used.

  Let's assume 100% redistribution. After the all loop is fully executed, the two arrays contain:

  ```
  updated_trove_yang_balances = [(A, 0), (B, 0)];
  new_yang_totals = [(B, 1000)];
  ```

  The final loop of the function is executed just once. Its first and only iteration writes the new total B value. However, it does not update the amount of B in the trove #1, since `(B, 0)` is the second element of the first array. The final state is that trove #1 still has 1000 units of B, but no more debt. The user U1 can now withdraw all 1000 units from the trove #1.

  This bug violates the shrine invariant "The total amount of a yang is equal to the sum of all troves' deposits of that yang (this includes any exceptionally redistributed yangs and their accompanying errors) and the initial amount seeded at the time of `add_yang`."

Proof of Concept

```
#[test]
fn test_shrine_redistribution_bug() {
    let shrine: IShrineDispatcher =
shrine_utils::shrine_setup_with_feed(Option::None)

    // Manually set up troves so that all troves
uses just yang1
    let yangs: Span<ContractAddress> =
shrine_utils::three_yang_addrs_reversed();
    let yang_addr = *yangs.at(1); // select the
middle one

    let forge_amt: u128 =
1_000_000_000_000_000_000_000;

    // Set up trove1 with some yang and some debt
    let trove1_owner =
common::trove1_owner_addr();
    let redistributed_trove: u64 =
common::TROVE_1;
    start_prank(CheatTarget::All,
shrine_utils::admin());
    shrine.deposit(yang_addr,
redistributed_trove,
shrine_utils::TROVE1_YANG1_DEPOSIT.into());
    shrine.forge(trove1_owner,
redistributed_trove, forge_amt.into(),
0_u128.into());

    // Set up trove1 with some yang and some debt
    let trove2_owner =
```

```
common::trove2_owner_addr();
    let recipient_trove: u64 = common::TROVE_2;
    shrine.deposit(yang_addr, recipient_trove,
shrine_utils::TROVE1_YANG1_DEPOSIT.into());
    shrine.forge(trove2_owner, recipient_trove,
forge_amt.into(), 0_u128.into());

    println!("before:");
    println!(" trove1 yang={}",
shrine.get_deposit(yang_addr,
redistributed_trove).val);
    println!(" trove2 yang={}",
shrine.get_deposit(yang_addr,
recipient_trove).val);
    println!(" total yang: {}",
shrine.get_yang_total(yang_addr));

    // Simulate complete redistribution of trove1
    shrine.redistribute(redistributed_trove,
trove1_health.debt, RAY_ONE.into());

    println!("after:");
    println!(" trove1 yang={}",
shrine.get_deposit(yang_addr,
redistributed_trove).val);
    println!(" trove2 yang={}",
shrine.get_deposit(yang_addr,
recipient_trove).val);
    println!(" total yang: {}",
shrine.get_yang_total(yang_addr));
```

```
shrine_utils::assert_shrine_invariants(shrine,
yangs, 2);
}
```

Output:

```
before:
trove1 yang=500000000000000000
trove2 yang=500000000000000000
trove1 value=2895610636113415002820
ltv=345350299355935952856010534
debt=100000000000000000000
trove2 value=2895610636113415002820
ltv=345350299355935952856010534
debt=100000000000000000000

after:
trove1 yang=500000000000000000
trove2 yang=500000000000000000
trove1 value=2895610636113415002820 ltv=0 debt=0
trove2 value=2895610636113415002820
ltv=690700598711871905712021068
debt=200000000000000000000
```

Expected output:

```
after:
trove1 yang=0
trove2 yang=500000000000000000
trove1 value=0 ltv=0 debt=0
trove2 value=2895610636113415002820
ltv=690700598711871905712021068
debt=200000000000000000000
```

One question is: how difficult is it to force the redistribution? It looks like it's a realistic option in some cases. For instance, the attacker could first drain the stability pool (absorber) by forcing absorbtions until it is empty. A liquidation can be forced by borrowing max amount and then waiting for `ltv > threshold` to happen due to small price fluctuations, potentially even manipulating the price slightly (as only a small change is required to cross the threshold). For a collateral asset with asset's `threshold > ABSORPTION_THRESHOLD` it's not required that the trove's `penalty == max_possible_penalty`.

Recommended Mitigation Steps

Do not update the array `updated_trove_yang_balances` before the `continue` statement.

Assessed type

Loop

tserg (Opus) confirmed and commented via duplicate issue #199**:**

> This is valid - potentially fixed.

0xsomeone (judge) commented**:**

> The warden has demonstrated how a debt redistribution will maintain incorrect entries in the updated Yang balances and total Yang balances when skipping over one, weaponizing this behavior to acquire collateral from a shrine.
>
> I believe a high-risk evaluation is apt as collateral of other users is directly impacted.

[H-04] Shrine's recovery mode can be weaponized as leverage to liquidate healthy troves

Submitted by 3docSec, also found by etherhood, nmirchev8, and kfx

Lines of code

https://github.com/code-423n4/2024-01-opus/blob/4720e9481a4fb20f4ab4140f9cc391a23ede3817/src/core/shrine.cairo#L1046

Description

In the Shrine implementation, the loan (trove) health is calculated by having its LTV compared to the shrine threshold:

```
File: shrine.cairo
1133:         fn is_healthy_helper(self:
@ContractState, health: Health) -> bool {
1134:             health.ltv <= health.threshold
1135:         }
---
1140:         fn assert_valid_trove_action(self:
@ContractState, trove_id: u64) {
1141:             let health: Health =
self.get_trove_health(trove_id);
1142:
assert(self.is_healthy_helper(health), 'SH: Trove
LTV is too high');
```

The shrine threshold is in turn calculated from the weighted thresholds of the yang deposits, scaled down by a variable factor, in case the shrine is in recovery mode:

```
File: shrine.cairo
1040:         fn get_trove_health(self:
@ContractState, trove_id: u64) -> Health {
```

```
---
1045:             let (mut threshold, mut value)
=
self.get_threshold_and_value(trove_yang_balances,
interval);
1046:             threshold =
self.scale_threshold_for_recovery_mode(threshold);
---
1202:         fn
scale_threshold_for_recovery_mode(self:
@ContractState, mut threshold: Ray) -> Ray {
1203:             let shrine_health: Health =
self.get_shrine_health();
1204:
1205:             if
self.is_recovery_mode_helper(shrine_health) {
1206:                 let
recovery_mode_threshold: Ray =
shrine_health.threshold *
RECOVERY_MODE_THRESHOLD_MULTIPLIER.into();
1207:                 return max(
1208:                     threshold *
THRESHOLD_DECREASE_FACTOR.into() *
(recovery_mode_threshold / shrine_health.ltv),
1209:                     (threshold.val /
2_u128).into()
1210:                 );
1211:             }
1212:
1213:             threshold
1214:         }
```

We can see from the above code that triggering recovery mode lowers the threshold, exposing the more under-collateralized loans (troves) to liquidation. This is expected behavior when the LTV fluctuations are coming from collateral price swings.

If we look at how recovery mode is triggered:

```
File: shrine.cairo
0079:     const
RECOVERY_MODE_THRESHOLD_MULTIPLIER: u128 =
700000000000000000000000000; // 0.7 (ray)
---
1165:         fn is_recovery_mode_helper(self:
@ContractState, health: Health) -> bool {
1166:             let recovery_mode_threshold:
Ray = health.threshold *
RECOVERY_MODE_THRESHOLD_MULTIPLIER.into();
1167:             health.ltv >=
recovery_mode_threshold
1168:         }
```

We can see that all it takes to trigger recovery mode is to bring the shrine LTV to 70% of its nominal threshold, or higher. This can be achieved by a malicious (or naive) user, provided they have enough collateral to take large borrows close to the collateralization threshold, and the shrine `debt_ceiling` provides enough headroom.

Impact

Loans can be forced into liquidation territory, and be liquidated, whenever a new loan is opened large enough to trigger recovery mode. This can also happen as a deliberate attack, and within a single transaction, without exposing the attacker's funds to

liquidation. It is consequently a solid candidate for a flash loan attack, but can also be executed with a large amount of pre-deposited collateral.

Proof of Concept

The following test case can be added to `test_shrine.cairo` to show how a large collateral injection + large loan can force a pre-existing loan into an unhealthy state, ready to be liquidated:

```
    #[test]
    fn test_shrine_recovery() {
        let wad: u128 = 10000000000000000;
        let shrine: IShrineDispatcher =
shrine_utils::shrine_setup_with_feed(Option::None)
        let yangs: Span<ContractAddress> =
shrine_utils::three_yang_addrs();
        let yang1_addr: ContractAddress =
*yangs.at(0);

        // trove 1: deposits 1 wad, mints nothing
- they just contribute to the health of the
protocol

start_prank(CheatTarget::One(shrine.contract_addre
shrine_utils::admin());
        let trove1_deposit: u128 = 1 * wad;
        shrine.deposit(yang1_addr, 1,
trove1_deposit.into());

        // trove 2: deposits 1 wad, mints 90% of
what they can (slightly overcollateralized)
```

```
        let trove2_deposit: u128 = 1 * wad;
        shrine.deposit(yang1_addr, 2,
trove2_deposit.into());

        let forge_amt2 =
shrine.get_max_forge(2).val * 9 / 10;
        shrine.forge(shrine_utils::admin(), 2,
forge_amt2.into(), WadZeroable::zero());

        // life is good
        let mut health =
shrine.get_shrine_health();
        assert(false ==
shrine.is_recovery_mode(), '');

        // trove 3: deposits a flash-loaned
collateral, mints A LOT to raise the LTV
        let trove3_deposit: u128 = 10 * wad;
        shrine.deposit(yang1_addr, 3,
trove3_deposit.into());

        let forge_amt3: u128 =
shrine.get_max_forge(3).val * 85 / 100;
        shrine.forge(shrine_utils::admin(), 3,
forge_amt3.into(), WadZeroable::zero());

        health = shrine.get_shrine_health();
        let trove2_health =
shrine.get_trove_health(2);

        // things are not good anymore. Shrine is
```

```
in recovery mode and trove 2 can now be
liquidated
        assert(shrine.is_recovery_mode(), '');
        assert(trove2_health.ltv >
trove2_health.threshold, '')
    }
```

Tools Used

Foundry

Recommended Mitigation Steps

It is not entirely clear how the recovery mechanism, intended as is, can be modified to fix this issue. Introducing a form of limitation to liquidations happening in the same block of a recovery trigger can mitigate exposure to flash-loans, but large loans against pre-owned collateral left dormant on the shrine would still be a viable attack path.

What we can tell, however, is that the recovery mechanism appears to have the intent of increasing the difficulty of opening new loans as the shrine health approaches the liquidation threshold.

Popular DeFi protocols like Compound solved this very issue by having two different LTV references: one for accepting liquidations and one lower for accepting new loans.

More in detail, the protocol is vulnerable only because one can borrow at LTV values above the recovery threshold (70% of the nominal threshold) but still below the liquidation threshold. Therefore, is able to raise the global LTV above that recovery threshold. If users were not allowed to borrow above that 70%, they wouldn’t be able to raise the global LTV above it, even with infinite collateral.

Assessed type

MEV

tserg (Opus) confirmed and commented via duplicate issue #205:

> This is valid - potentially fixed.

0xsomeone (judge) commented:

> The warden has demonstrated how the automatic recovery mode mechanism of the Opus system can be exploited to force the system into recovery mode, enabling the liquidation of previously healthy troves.
>
> A high-risk vulnerability rating for this issue is valid as the automatic recovery mode can be exploited within a single transaction to force the system into recovery mode by opening a bad position, liquidating whichever troves are lucrative, and closing the previously bad position with zero risk.

Medium Risk Findings (9)

[M-01] Loss of liquidation compensation assets in absorb

Submitted by etherhood

In absorb, since `shrine.melt` is being called after `free`, caller will end up receiving less compensation than actually intended. It is because in `free`, while checking user deposits, there are two cases when user has pending redistribution with exception: 1: User's yang 1 has zero balance 2: User's yang 1 has non zero balance

In both cases user's balances are due for an update, if first n yang balances are zero, caller will miss out on compensation from

unaccounted yang updates from all yangs due for redistribution until it encounters one with non zero yang balance. When it encounters first non-zero yang balance, it will still miss out on unaccounted fund from that yang as well, but after that it will be okay, since `charge()` would be called inside seize function which will then update balances for all yangs.

Proof of Concept

This is inside free function, where only after first non-zero yang balance is seize called and redistribution updated for trove and all its yangs. All calls to deposit will return less than actual balance of trove.

```
            loop {
                match yangs_copy.pop_front() {
                    Option::Some(yang) => {
                        let deposited_yang_amt:
Wad = shrine.get_deposit(*yang, trove_id);

                        let freed_asset_amt: u128
= if deposited_yang_amt.is_zero() {
                            0
                        } else {
                            let freed_yang: Wad =
wadray::rmul_wr(deposited_yang_amt,
percentage_freed);
                            let exit_amt: u128 =
sentinel.exit(*yang, recipient, trove_id,
freed_yang);
                            shrine.seize(*yang,
trove_id, freed_yang);
```

```
                            exit_amt
                        };



freed_assets.append(AssetBalance { address:
*yang, amount: freed_asset_amt });
                    },
                    Option::None => { break; }
                };
            };
```

Tools Used

VS Code

Recommended Mitigation Steps

The `absorb` function should update order of melt and free:

```
shrine.melt(absorber.contract_address, trove_id,
purge_amt);
            let compensation_assets:
Span<AssetBalance> = self.free(shrine, trove_id,
pct_value_to_compensate, caller);
```

tserg (Opus) commented:

> This is valid and mitigated.

tserg (Opus) confirmed, but disagreed with severity and commented:

> Caller still gets compensation, plus it is capped to 50 USD, so the impact of this is capped to 50 USD. User funds are also not at risk.

0xsomeone (judge) decreased severity to Medium and commented:

> The warden has showcased a way in which the compensation for the caller of the absorption will be lower than intended.
>
> While the issue is valid, I agree with the sponsor in that the severity of this cannot be considered high-risk. User funds are indeed affected, however, so I believe a medium-risk rating is more apt.

[M-02] after shut, no pulled redistribution yang will be locked

Submitted by bin2chen, also found by mahdikarimi and etherhood

In `caretaker.release()`, we can release the remaining yang:

```
        fn release(ref self: ContractState,
trove_id: u64) -> Span<AssetBalance> {
            let shrine: IShrineDispatcher =
self.shrine.read();
...
            loop {
                match yangs_copy.pop_front() {
                    Option::Some(yang) => {
@>                      let deposited_yang: Wad =
shrine.get_deposit(*yang, trove_id);
                        let asset_amt: u128 = if
deposited_yang.is_zero() {
                            0
                        } else {
                            let exit_amt: u128 =
sentinel.exit(*yang, trove_owner, trove_id,
deposited_yang);
                            // Seize the
collateral only after assets have been
```

```
                                // transferred so
that the asset amount per yang in Gate
                                // does not change
and user receives the correct amount
                                shrine.seize(*yang,
trove_id, deposited_yang);
                                exit_amt
                            };

released_assets.append(AssetBalance { address:
*yang, amount: asset_amt });
                        },
                        Option::None => { break; },
                    };
            };
```

As above, we can only release the yang that already exists in the trove: `shrine.get_deposit(*yang, trove_id);`.

When `shire.get_live() == false`, we can no longer perform `shire.pull_redistributed_debt_and_yangs()`. So if there is any yang that hasn't been pulled, it can't be retrieved through `caretaker.release()`. This part of the yang will be locked in the contract.

Impact

After `shut()`, the `redistributed yang` that hasn't been pulled will be locked in the contract.

Recommended Mitigation

Add `pull_when_shut()`:

```
        fn pull_when_shut(ref self:
ContractState, trove_id: u64) {
            if self.is_live.read() {
                return;
            }
           let trove: Trove =
self.troves.read(trove_id);

            let current_interval: u64 = now();
            let trove_yang_balances:
Span<YangBalance> =
self.get_trove_deposits(trove_id);
            let (updated_trove_yang_balances, _)
= self

.pull_redistributed_debt_and_yangs(trove_id,
trove_yang_balances,Wad { val: 0 });

            // If there was any exceptional
redistribution, write updated yang amounts to
trove
            if
updated_trove_yang_balances.is_some() {
                let mut
updated_trove_yang_balances =
updated_trove_yang_balances.unwrap();
                loop {
                    match
updated_trove_yang_balances.pop_front() {

Option::Some(yang_balance) => {
```

```
self.deposits.write((*yang_balance.yang_id,
trove_id), *yang_balance.amount);
                        },
                        Option::None => { break;
},
                    };
                };
            }

self.trove_redistribution_id.write(trove_id,
self.redistributions_count.read());
        }
```

```
        fn release(ref self: ContractState,
trove_id: u64) -> Span<AssetBalance> {
            let shrine: IShrineDispatcher =
self.shrine.read();
...
+           shrine.pull_when_shut(trove_id);
            loop {
                match yangs_copy.pop_front() {
                    Option::Some(yang) => {
                        let deposited_yang: Wad =
shrine.get_deposit(*yang, trove_id);
```

Assessed type

Error

tserg (Opus) confirmed

0xsomeone (judge) commented:

> The warden has demonstrated how the graceful shutdown of a

> shrine can result in loss of funds if an exceptional trove redistribution occurred, resulting in loss of the redistributed collateral.
>
> I believe a medium-risk grade is apt for this exhibit as it relates to a low-likelihood scenario and a loss that is contained.

[M-03] ERC4626 inflate issue mitigation is not sufficient

Submitted by jasonxiale, also found by bin2chen

Lines of code

https://github.com/code-423n4/2024-01-opus/blob/4720e9481a4fb20f4ab4140f9cc391a23ede3817/src/core/gate.cairo#L191-L223
https://github.com/code-423n4/2024-01-opus/blob/4720e9481a4fb20f4ab4140f9cc391a23ede3817/src/core/absorber.cairo#L667-L705

Impact

Both `absorber` and `gate` use the same mitigation for ERC4626 first depositor front-running vulnerability, but current implementation is not sufficient. By abusing the flaw, even though malicious attacker can't benefit from the mitigation, he can cause other normal users to lose assets.

Proof of Concept

Because of `absorber` and `gate` use the same mitigation, I will take `gate` as example.

Suppose the yang's decimals is **18**:

When `sentinel.add_yang` is called to add yang to shrine, `initial_yang_amt` is passed to `shrine.add_yang` as mitigation to the inflate issue. And `initial_yang_amt` is set as `INITIAL_DEPOSIT_AMT` which is `const INITIAL_DEPOSIT_AMT: u128 = 1000;`.

In sentinel.cairo#L204-L206, `yang_erc20.transfer_from` is called to transfer 1000 wei `yang_erc` from caller to gate.

And then the code flow will fall into `shrine.add_yang`. When the function is called, `initial_yang_amt` is still `1000`.

In `shrine.add_yang`, the `yang_total` will be set to `1000` in shrine.cairo#L591.

So when the admin (which is the first depositor) calls `sentinel.add_yang`, he will transfer 1000 wei `yang_asset` and will receive 1000 `yang_amt` yang.

After that, when the second user calls `abbot.open_trove`, `gate.convert_to_yang_helper` will calculate the `yang_amt` by gate.cairo#L220:

```
        fn convert_to_yang_helper(self:
@ContractState, asset_amt: u128) -> Wad {
            let asset: IERC20Dispatcher =
self.asset.read();
            let total_yang: Wad =
self.get_total_yang_helper(asset.contract_address)

            if total_yang.is_zero() {
                let decimals: u8 =
asset.decimals();
                // Otherwise, scale `asset_amt`
up by the difference to match `Wad`
```

```
                // precision of yang. If asset is
of `Wad` precision, then the same
                // value is returned
                fixed_point_to_wad(asset_amt,
decimals)
            } else {
                (asset_amt.into() * total_yang) /
get_total_assets_helper(asset).into() <<<--- the
second user calcuates the `yang_amout` using this
code
            }
        }
```

And gate.get_total_assets_helper is using asset.balance_of:

```
    #[inline(always)]
    fn get_total_assets_helper(asset:
IERC20Dispatcher) -> u128 {

asset.balance_of(get_contract_address()).try_into(
<<<--- balance_of is used here
    }
```

So to sum up, in the current implementation:

1. The first depositor (which is also the admin) will deposit 1000 wei ERC, and will receive 1000 yang_amt.
2. The second depositor will get his yang_amt based off (asset_amt.into() * total_yang) / get_total_assets_helper(asset).into().

   Suppose in a case that the asset is DAI (18 decimals):

1. When yang_dai is created, the admin calls add_yang to add DAI

into the protocol, and he will transfer 1000 wei DAI to `gate`, and will receive 1000 `yang_amt`.

2. A malicious tranfers (`100000 * 1e18 - 1000` wei) DAI (which is worth 100000 USD) to `gate`.
3. Alice, a normal user, deposits `199 * 1e18` DAI (which is worth 199 USD) by calling `abbot.open_trove`, based on gate.cairo#L220. Alice will get `199 * 1e18 * 1000 / (100000 * 1e18)` = 1 wei.
4. If Alice wants to close her trove, `gate.convert_to_assets` will be used to calculate the asset amount, according to gate.cairo#L191-L204.

   gate.cairo#L202 will be used: `((yang_amt * get_total_assets_helper(asset).into()) / total_yang).val`, because Alice has 1 wei `yang_amout`, and `get_total_assets_helper()` is `(100000 + 199) * 1e18` and `total_yang` will be 1001.

   So Alice will get `1 * (100000 * 1e18 + 199 * 1e18) / (1000 + 1) = 100.0989010989011 * 1e18`, which is 100 USD. And the remaining `99 * 1e18` DAI will be in the gate.

   In the above case, there will be two issues:

1. Alice will lose her assets.
2. After Alice closes her trove, the system's balance will be `(100000 + 99) *1e18`, which is becoming larger. And the balance will keep growing with time going, which will cause users to lose more assets.

Tools Used

VIM

Recommended Mitigation Steps

In `gate.get_total_assets_helper`, don't use `balance_of` to calculate the amount; instead, define a new variables and record the deposited asset amount by the variables

Assessed type

ERC4626

tserg (Opus) confirmed via duplicate issue #196

0xsomeone (judge) commented:

> This submission details the vulnerability with a greater impact than #196, and has thus been selected as primary.

[M-04] The `provide()` function does not reset withdrawal requests, allowing an attacker to bypass risk-free yield tactics protection

Submitted by minhquanym

In the Opus protocol, the absorber is a stability pool that permits yin holders to contribute their yin and participate in liquidations (also known as absorptions) as a consolidated pool. Provided yin from users could be absorbed during an absorption. In return, users receive yang, which usually has a higher value than the absorbed yin. However, in the event of bad debt, it could be less. This situation could lead to risk-free yield frontrunning tactics, where an attacker only provides yin when the absorption brings profit then removing all yin right after that.

The Opus team is aware of this attack vector and has, therefore, implemented a request-withdraw procedure. Users must first request a withdrawal and wait for a certain period before executing

the withdrawal.

However, the `provide()` method does not reset these request timestamps, which allows an attacker to bypass this safeguard.

Proof of Concept

The attacker's tactics will be:

1. Deposit a small amount of yin into 100 of his accounts. Note that the total value of these small amounts is negligible. Also please note that the value `100` is just arbitrary value to describe the idea. It could be higher or lower depending on the `REQUEST_BASE_TIMELOCK` and `REQUEST_VALIDITY_PERIOD`.
2. Regularly request withdrawals for these accounts to ensure that at least one account has a valid request at any given time. This can be easily achieved by requesting a withdrawal from the first account at timestamp `X`, the second at `X + REQUEST_VALIDITY_PERIOD`, the third at `X + 2 * REQUEST_VALIDITY_PERIOD`, and so on. The first account request will expire at `X + REQUEST_BASE_TIMELOCK + REQUEST_VALIDITY_PERIOD`, but now second request account becomes valid because it was requested at `X + REQUEST_VALIDITY_PERIOD`.
3. Since the `provide()` function does not reset the request, the attacker can simply select one account with a valid removing request and perform the "risk-free yield tactics". He does this by providing a large amount of yin to this account to earn yield and then immediately calls `remove()`.

Recommended Mitigation Steps

Reset the withdrawal request in the `provide()` function.

Assessed type

MEV

tserg (Opus) confirmed, but disagreed with severity and commented:

> The economic feasibility of the exploit is questionable, and it requires extensive infrastructure to be set up and maintained. Ultimately, no user funds are at risk.

0xsomeone (judge) decreased severity to Medium and commented:

> The warden has demonstrated how a malicious user can game the queued withdrawal system of Opus and cycle withdrawal authorizations between multiple accounts to always retain one that can immediately withdraw at any given point in time. As a provision will not reset those requests, the account whose withdrawal authorization is valid at a point a lucrative absorption occurs can be exploited to acquire a bigger share of the collateral.
>
> I believe a medium-risk grade is better suited for this finding as only the “reward” distribution is manipulated and it requires extensive money-translated effort (i.e. gas for maintaining positions active) that will further reduce the economic viability of this exploit.

Note: For full discussion, see here.

[M-05] An attacker could manipulate debt exceptional redistribution because it is allowed to deposit into any trove

Submitted by minhquanym

In a redistribution, the unhealthy trove’s collateral and debt is distributed among troves proportionally to its collateral composition.

If no other troves have deposited a yang that is to be redistributed, then the debt and value attributed to that yang will be redistributed among all other yangs in the system, according to their value proportionally to the total value of all remaining yangs in the system.

However, the Opus protocol allows anyone to deposit into any trove, even if they are not the trove's owner, as seen in the function `abbot.deposit()`. This feature could potentially be exploited by an attacker. For instance, if a yang is only deposited in one trove that's being redistributed, the attacker could deposit a small amount into a victim's trove. This could result in all bad debt being redistributed to the victim's trove instead of exceptional redistribution, even if the victim didn't want this (i.e., they didn't deposit this yang into their trove).

```
fn deposit(ref self: ContractState, trove_id:
u64, yang_asset: AssetBalance) {
    // There is no need to check the yang address
is non-zero because the
    // Sentinel does not allow a zero address
yang to be added.

    assert(trove_id != 0, 'ABB: Trove ID cannot
be 0');
    assert(trove_id <= self.troves_count.read(),
'ABB: Non-existent trove');
    // note that caller does not need to be the
trove's owner to deposit
    // @audit Attacker could deposit for any
trove to make it the only trove deposited this
yang and being redistributed when bad debt happen
```

```
    self.deposit_helper(trove_id,
get_caller_address(), yang_asset);
}
```

Proof of Concept

Consider the following scenario:

1. Assume there is a highly volatile yang X added to Opus. Alice, aware of the risk of X, does not deposit any amount of X into her trove.
2. Bob, the attacker, has deposited some X into his trove.
3. When Bob's trove is about to be redistributed due to bad debt, since no other trove has deposited X, the debt should be redistributed among all other yangs (exceptional redistribution). This means everyone would share the loss of the bad debt. However, Bob could bypass this by depositing some X into Alice's trove. Since the deposit function doesn't require the caller to be the trove's owner, Alice's trove now holds some X. As a result, a regular redistribution will occur, and all of Bob's bad debt will be redistributed to Alice's trove.

Recommended Mitigation Steps

Limit deposits to only the trove's owner.

tserg (Opus) acknowledged, but disagreed with severity and commented**:**

> The likelihood of this happening is low.

0xsomeone (judge) decreased severity to Medium and commented**:**

> The warden has demonstrated how bad debt re-allocation can be "gamed" due to the permissionless deposit system of troves.

> This particular attack vector is quite interesting and I commend the warden for their out-of-the-box thinking. However, I believe a medium-risk grade is better suited, given that the likelihood of a liquidation of a trove with an asset that is not held by any other trove in the system is low.

[M-06] Multiplier is incorrectly calculated in `Controller`

Submitted by ABAIKUNANBAEV

In `Controller` smart contract, due to mistake in multiplication of `i_gain`, it's miscalculated and the error occurs. This can lead to a multiplier reflecting an incorrect value and therefore affect yin borrowing rate.

Proof of Concept

According to the multiplier formula, first we calculate `p_term`. The formula for `p_term` is following:

`p_term = p_gain * error`

And that's done right using the `get_p_term_internal()` function:

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/controller.cairo#L243-245

```
fn get_p_term_internal(self: @ContractState) ->
SignedRay {
            self.p_gain.read() *
nonlinear_transform(self.get_current_error(),
self.alpha_p.read(), self.beta_p.read())
        }
```

Then we add right and go to integral term calculation. It's almost the same as `p_term`, only the time scale factor is added:

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/controller.cairo#L256-258

```
  old_i_term
                +
nonlinear_transform(self.get_prev_error(),
self.alpha_i.read(), self.beta_i.read())
                    *
time_since_last_update_scaled
```

The problem is that when we add this formula to the multiplier, we multiply it additionally by `i_gain`:

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/controller.cairo#L166-167

```
 let new_i_term: SignedRay =
self.get_i_term_internal();
                multiplier += i_gain *
new_i_term;
```

And, according to the formula, only this part should be multiplied by `i_gain`:

```
nonlinear_transform(self.get_prev_error(),
self.alpha_i.read(), self.beta_i.read())
                    *
time_since_last_update_scaled
```

But `old_term` also ends up multiplied.

Recommended Mitigation Steps

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/

controller.cairo#L256-258

```
-                old_i_term +
nonlinear_transform(self.get_prev_error(),
-                self.alpha_i.read(),
self.beta_i.read()) *
-                time_since_last_update_scaled

+                old_i_term + i_gain *
nonlinear_transform(self.get_prev_error(),
+                self.alpha_i.read(),
self.beta_i.read()) *
+                time_since_last_update_scaled
```

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/controller.cairo#L167

```
-                multiplier += i_gain *
new_i_term;
+                multiplier += new_i_term;
```

0xsomeone (judge) commented:

> The warden has demonstrated how the formula that is in use by the Controller contradicts the documentation of the project, and namely the specification of the aforementioned formula.
>
> In detail, the specification states:
>
> *Note: Please see provided formula in the judge's original comment.*
>
> We are interested in the latter part of the formula, specifically:
>
> *Note: Please see provided formula in the judge's original comment.*
>
> The documentation states that `k_i` stands for the:
>
> > gain that is applied to the integral term

The problem highlighted by the warden is that the implementation will calculate the following:

Note: Please see provided formula in the judge's original comment.

The above corresponds to:

```
let new_i_term: SignedRay =
self.get_i_term_internal();
multiplier += i_gain * new_i_term;
```

Whereby the `get_i_term_internal` function will ultimately yield:

```
old_i_term
    + nonlinear_transform(self.get_prev_error(),
self.alpha_i.read(), self.beta_i.read())
    * time_since_last_update_scaled
```

In the above:

Note: please review provided terms in the judge's original comment.

Based on the above analysis, the documentation indeed contradicts what the code calculates. The Opus team is invited to evaluate this exhibit; however, regardless of the correct behaviour of the system, this submission will be accepted as valid due to the documentation being the "source of truth" during the audit's duration.

milancermak (Opus) confirmed

[M-07] Collateral cannot be withdrawn from trove once yang is suspended

Submitted by 0xTheC0der

Once a yang (collateral asset) is suspended on Opus, the following holds true according to the documentation:

> No further deposits can be made. This is enforced by the Sentinel.
>
> Its threshold will decrease to zero linearly over the `SUSPENSION_GRACE_PERIOD`.

Therefore, a user will naturally deposit healthier collateral to their trove to maintain its threshold, while withdrawing the unhealthy/ suspended collateral as it loses value on Opus, effectively replacing the collateral.

However, this is not possible because the underlying assets of a yang are immediately frozen once suspended. It's clear from the documentation that no more deposits of suspended collateral can be made, but this *accidentally* also affects the withdrawals.

The `abbot::withdraw(...)` method calls `sentinel::convert_to_yang(...)` which in turn calls `sentinel::assert_can_enter(...)`:

```
fn assert_can_enter(self: @ContractState, yang:
ContractAddress, gate: IGateDispatcher,
enter_amt: u128) {
    ...

    let suspension_status: YangSuspensionStatus =
self.shrine.read().get_yang_suspension_status(yang
    assert(suspension_status ==
YangSuspensionStatus::None, 'SE: Yang
suspended');

    ...
}
```

One can see that the assertion will be triggered once the suspension status is `Temporary` or `Permanent`.

As a consequence, a yang's underlying assets are frozen once suspended:

- If the suspension status is still `Temporary`, the admin can unsuspend the yang to make it withdrawable again. However, this also stops its threshold decrease and makes it depositable again, which eliminates the incentives to withdraw in the first place. This essentially breaks the suspension mechanism.
- If the suspension status reaches `Permanent`, the assets are permanently locked from withdrawal. Nevertheless, there is a workaround by closing the trove, which requires all the yin (debt) to be repaid and unnecessarily withdraws all other yangs (collateral assets) of the trove too. Therefore, this is not a viable solution for the present issue.

Proof of Concept

The following PoC demonstrates that a yang's underlying assets cannot be withdrawn once suspended.

Add the *test case* below to `src/tests/abbot/test_abbot.cairo` and run it with `snforge test test_withdraw_suspended_fail`.

```
#[test]
#[should_panic(expected: ('SE: Yang suspended',))]
fn test_withdraw_suspended_fail() {
    let (shrine, sentinel, abbot, yangs, _, trove_owner, trove_id, _, _) = abbot_utils::deploy_abbot_and_open_trove(
```

```
        Option::None, Option::None, Option::None,
Option::None, Option::None
    );
    let asset_addr: ContractAddress =
*yangs.at(0);
    let amount: u128 = WAD_SCALE;


start_prank(CheatTarget::One(sentinel.contract_add
sentinel_utils::admin());
    sentinel.suspend_yang(asset_addr);

stop_prank(CheatTarget::One(sentinel.contract_addr


start_prank(CheatTarget::One(abbot.contract_addres
trove_owner);
    abbot.withdraw(trove_id, AssetBalance {
address: asset_addr, amount });
}
```

Recommended Mitigation Steps

Replace the accidental `assert_can_enter(..)` check with those that are really necessary at this point:

```
diff --git a/src/core/sentinel.cairo b/src/core/
sentinel.cairo
index b18edde..9671ca2 100644
--- a/src/core/sentinel.cairo
+++ b/src/core/sentinel.cairo
@@ -156,7 +156,8 @@ mod sentinel {
```

```
        // This can be used to simulate the
effects of `enter`.
        fn convert_to_yang(self: @ContractState,
yang: ContractAddress, asset_amt: u128) -> Wad {
            let gate: IGateDispatcher =
self.yang_to_gate.read(yang);
-           self.assert_can_enter(yang, gate,
asset_amt);
+
assert(gate.contract_address.is_non_zero(), 'SE:
Yang not added'); // alike to
sentinel::convert_to_assets(...)
+           assert(self.yang_is_live.read(yang),
'SE: Gate is not live');      // to satisfy
test_sentinel::test_kill_gate_and_preview_enter()
            gate.convert_to_yang(asset_amt)
        }
```

Assessed type

Invalid Validation

tserg (Opus) confirmed

0xsomeone (judge) commented:

> The warden has demonstrated how a Yang's suspension (either temporary or permanent) will cause the overall system to deviate from its specification and disallow withdrawals of the Yang (collateral) instead of permitting them, thereby never letting the system gradually recover.
>
> I consider a medium-risk severity to be appropriate for this exhibit as it relates to misbehavior that will arise during an emergency scenario.

[M-08] Attacker can lock every trove withdrawals

Submitted by [zigtur]

Lines of code

https://github.com/code-423n4/2024-01-opus/blob/main/src/core/abbot.cairo#L210
https://github.com/code-423n4/2024-01-opus/blob/main/src/core/sentinel.cairo#L159
https://github.com/code-423n4/2024-01-opus/blob/main/src/core/sentinel.cairo#L288

Description

During `withdraw`, the `convert_to_yang` function of sentinel is called. This function converts an asset amount to a Yang amount.

`convert_to_yang` then calls `assert_can_enter` to ensure that `current_total + enter_amt <= max_amt`. This check is incorrect in the case of a withdrawal, as a subtraction should be made instead of an addition.

An attacker can deposit an asset amount such that `current_total == max_amt`. In such condition, withdrawing assets will be impossible because of the incorrect check.

Impact

In every Gate for which a maximum amount of asset is set in Sentinel, an attacker can lock asset withdrawals for every users.

Proof of Concept

The following test can be added to `src/tests/abbot/`

`test_abbot.cairo` to show:

```
    #[test]
    #[should_panic(expected: ('SE: Exceeds max
amount allowed',))]
    fn
test_zigtur_exploit_withdraw_exceeds_asset_cap_fai
{
        let (_, sentinel, abbot, yangs, gates,
trove_owner, trove_id, _, _) =
abbot_utils::deploy_abbot_and_open_trove(
            Option::None, Option::None,
Option::None, Option::None, Option::None
        );

        let asset_addr: ContractAddress =
*yangs.at(0);
        let gate_addr: ContractAddress =
*gates.at(0).contract_address;
        let gate_bal = IERC20Dispatcher {
contract_address: asset_addr
}.balance_of(gate_addr);

        // Set the maximum to current value,
which means `current_total == max_amt`

start_prank(CheatTarget::One(sentinel.contract_add
sentinel_utils::admin());
        let new_asset_max: u128 =
gate_bal.try_into().unwrap();
        sentinel.set_yang_asset_max(asset_addr,
new_asset_max);
```

```
stop_prank(CheatTarget::One(sentinel.contract_addr

        // User withdraws, so the max should not
be reached.
        // But `assert_can_enter()` in sentinel
uses addition even if withdrawing, so it reverts.
        let amount: u128 = 1;

start_prank(CheatTarget::One(abbot.contract_addres
trove_owner);
        abbot.withdraw(trove_id, AssetBalance {
address: asset_addr, amount });
    }
```

Note: The PoC doesn't show the attacker adding Yang asset to make withdrawals impossible. It only shows that the withdrawals are impossible, but the attack vector is clearly identifiable.

Tools Used

Unit testing

Recommended Mitigation Steps

During withdrawals, a `withdraw_to_yang()` function could be used instead of `convert_to_yang()`. This new function would not call `assert_can_enter`.

The following patch fixes this issue by implementing a `withdraw_to_yang` function, which calculates the amount of Yang without reverting:

```
diff --git a/src/core/abbot.cairo b/src/core/
```

```
abbot.cairo
index 1f0a589..1ededa4 100644
--- a/src/core/abbot.cairo
+++ b/src/core/abbot.cairo
@@ -207,7 +207,7 @@ mod abbot {
             let user = get_caller_address();
             self.assert_trove_owner(user,
trove_id);

-            let yang_amt: Wad =
self.sentinel.read().convert_to_yang(yang_asset.ad
yang_asset.amount);
+            let yang_amt: Wad =
self.sentinel.read().withdraw_to_yang(yang_asset.a
yang_asset.amount);
             self.withdraw_helper(trove_id, user,
yang_asset.address, yang_amt);
         }

diff --git a/src/core/sentinel.cairo b/src/core/
sentinel.cairo
index b18edde..644bb85 100644
--- a/src/core/sentinel.cairo
+++ b/src/core/sentinel.cairo
@@ -160,6 +160,15 @@ mod sentinel {
             gate.convert_to_yang(asset_amt)
         }

+        fn withdraw_to_yang(self:
@ContractState, yang: ContractAddress, asset_amt:
u128) -> Wad {
```

```
+            let gate: IGateDispatcher =
self.yang_to_gate.read(yang);
+
assert(gate.contract_address.is_non_zero(), 'SE:
Yang not added');
+            assert(self.yang_is_live.read(yang),
'SE: Gate is not live');
+            let suspension_status:
YangSuspensionStatus =
self.shrine.read().get_yang_suspension_status(yang
+            assert(suspension_status ==
YangSuspensionStatus::None, 'SE: Yang
suspended');
+            gate.convert_to_yang(asset_amt)
+        }
+
         // This can be used to simulate the
effects of `exit`.
         fn convert_to_assets(self:
@ContractState, yang: ContractAddress, yang_amt:
Wad) -> u128 {
             let gate: IGateDispatcher =
self.yang_to_gate.read(yang);
diff --git a/src/interfaces/ISentinel.cairo b/
src/interfaces/ISentinel.cairo
index 4149a38..04d0d04 100644
--- a/src/interfaces/ISentinel.cairo
+++ b/src/interfaces/ISentinel.cairo
@@ -33,5 +33,6 @@ trait ISentinel<TContractState>
{
     fn unsuspend_yang(ref self: TContractState,
```

```
yang: ContractAddress);
     // view
     fn convert_to_yang(self: @TContractState,
yang: ContractAddress, asset_amt: u128) -> Wad;
+    fn withdraw_to_yang(self: @TContractState,
yang: ContractAddress, asset_amt: u128) -> Wad;
     fn convert_to_assets(self: @TContractState,
yang: ContractAddress, yang_amt: Wad) -> u128;
 }
```

Note: Unit tests are all passing with the fix. To apply the patch, import the content in a `fix.patch` file, then execute `git apply fix.patch`.

Assessed type

DoS

0xsomeone (judge) decreased severity to Medium and commented:

> The warden has demonstrated how the withdrawal flow in the Abbot contract an incorrect conversion will occur that will enforce a deposit limitation assuming the withdrawn amount is newly deposited.
>
> This behavior will result in the withdrawal not going through and its equality case can be exploited to force all positions of the Yang to not be withdrawable until the limit is updated. I consider a medium-risk better suited for this submission as:
>
> - The funds will not be permanently lost and a reconfiguration can recover them.
> - The attacker would have sacrificed their funds (as they wouldn't be able to withdraw either), and those funds would be substantial.

> It is, however, a flaw that needs to be remediated by the Opus team.

milancermak (Opus) confirmed and commented:

> We mitigated this issue based on this report. Thanks!

[M-09] Unhealthy troves with LTV > 90% cannot always be absorbed as intended

Submitted by 0xTheC0der

Unhealthy troves with `ltv > 90%` and `threshold < 90%` cannot always be absorbed due to a wrong if-condition. According to Priority of liquidation methods it should always be possible to absorb unhealthy troves with `ltv > 90%`:

> Absorption can happen only after an unhealthy trove's LTV has exceeded the LTV at which the maximum possible penalty is reached, or if it has exceeded 90% LTV. The liquidation penalty in this case will similarly be capped to the maximum of 12.5% or the maximum possible penalty.

However, the `purger::get_absorption_penalty_internal(...)` method mistakenly checks the `threshold` instead of the `ltv` against the `ABSORPTION_THRESHOLD (90%)` in L467:

```
fn get_absorption_penalty_internal(
    self: @ContractState, threshold: Ray, ltv:
Ray, ltv_after_compensation: Ray
) -> Option<Ray> {
    if ltv <= threshold {
        return Option::None;
    }
```

```
    ...

    let mut max_possible_penalty: Ray = min(
        (RAY_ONE.into() - ltv_after_compensation)
/ ltv_after_compensation, MAX_PENALTY.into()
    );

    if threshold > ABSORPTION_THRESHOLD.into() {
// @audit ltv instead
        let s = self.penalty_scalar.read();
        let penalty = min(MIN_PENALTY.into() + s
* ltv / threshold - RAY_ONE.into(),
max_possible_penalty);

        return Option::Some(penalty);
    }

    let penalty = min(MIN_PENALTY.into() + ltv /
threshold - RAY_ONE.into(),
max_possible_penalty);

    if penalty == max_possible_penalty {
        Option::Some(penalty)
    } else {
        Option::None
    }
}
```

As a consequence, unhealthy troves can only be absorbed if they reach the maximum possible penalty although the condition `ltv > 90%` is already satisfied. This is against the protocol's intended

liquidation/absorption incentives and therefore, endangers the solvency of the protocol.

By observing the sponsor's graph for liquidation penalty it becomes evident that the MAX_PENALTY can only be achieved for `ltv` up to 89%. For even higher `ltv` up to `100%`, the penalty approaches `0%` due to `max_possible_penalty` (see code above), which lowers the incentives for liquidation and makes absorption a necessity.

In case of `threshold > 83%` there is a window where `90% < ltv < ltv@max_possible_penalty` causing absorptions to be impossible due to the present bug. This linked graph visualizes the present issue.

Proof of Concept

In the following, a numerical example is provided to demonstrate the above claims. Initial assumptions:

```
threshold              = 88%    (reasonable
because shrine::MAX_THRESHOLD is 100%)
ltv                    = 91%    (should be
eligible for absorption in any case)
ltv_after_compensation = 93%    (reasonable
because ltv becomes worse due to compensation)
```

Let's do the math step-by-step:

```
fn get_absorption_penalty_internal(
    self: @ContractState, threshold: Ray, ltv:
Ray, ltv_after_compensation: Ray
) -> Option<Ray> {
    // @PoC: 91% <= 88% --> false, skip body
    if ltv <= threshold {
        return Option::None;
```

```
    }

    ...

    let mut max_possible_penalty: Ray = min(
        (RAY_ONE.into() - ltv_after_compensation)
/ ltv_after_compensation, MAX_PENALTY.into()
    );
    // @PoC: max_possible_penalty = 7.53%

    // @PoC: 88% > 90% --> false, skip body due
to wrong check
    if threshold > ABSORPTION_THRESHOLD.into() {
// @audit ltv instead
        let s = self.penalty_scalar.read();
        let penalty = min(MIN_PENALTY.into() + s
* ltv / threshold - RAY_ONE.into(),
max_possible_penalty);

        return Option::Some(penalty);
    }

    let penalty = min(MIN_PENALTY.into() + ltv /
threshold - RAY_ONE.into(),
max_possible_penalty);
    // @PoC: penalty = 6.41%

    // @PoC: 6.41% == 7.53% --> false, go in else
    if penalty == max_possible_penalty {
        Option::Some(penalty)
    } else {
```

```
        Option::None    // @PoC: trove is not
eligible for absorption
    }
}
```

We can see that the trove has not reached its maximum possible penalty yet, therefore, it cannot be absorbed as expected, although `ltv > 90%`.

Recommended Mitigation Steps

Make sure the absorption threshold is checked against the `ltv` as intended:

```
diff --git a/src/core/purger.cairo b/src/core/
purger.cairo
index 6a36bbc..820aff5 100644
--- a/src/core/purger.cairo
+++ b/src/core/purger.cairo
@@ -464,7 +464,7 @@ mod purger {
                (RAY_ONE.into() -
ltv_after_compensation) / ltv_after_compensation,
MAX_PENALTY.into()
            );

-           if threshold >
ABSORPTION_THRESHOLD.into() {
+           if ltv > ABSORPTION_THRESHOLD.into()
{
                let s =
self.penalty_scalar.read();
                let penalty =
min(MIN_PENALTY.into() + s * ltv / threshold -
```

```
RAY_ONE.into(), max_possible_penalty);
```

Assessed type

Math

0xsomeone (judge) commented:

> The warden has demonstrated how a contradiction between the documentation and the implementation of the project will cause certain troves to not be liquidate-able temporarily.
>
> I confirmed this submission as the documentation of the project states in the priority of liquidation methods chapter that a trove should be liquidate-able if its TVL exceeds 90% (i.e. the `ABSORPTION_THRESHOLD`). The code incorrectly validates the trove's threshold rather than LTV, rendering the submission to be valid.
>
> I consider a medium-risk severity apt for this finding as the DoS is temporary.

tserg (Opus) commented:

> This is an error in the documentation.
>
> The correct wording should be:
>
> ```
> Absorption can happen only after an unhealthy
> trove's LTV has exceeded the LTV at which the
> maximum possible penalty is reached, or if its
> threshold exceeds 90% LTV and its LTV has
> exceeded the threshold
> ```

0xsomeone (judge) commented:

> The sponsor has clarified that the documentation was incorrect and that the code behaves as expected; however, per C4 standards I will accept this submission as valid given that the documentation

> serves as the source of truth for the wardens to validate.
>
> 0xTheC0der (warden) commented:
>
> > First of all, thanks for keeping the issue valid due to the *source of truth* consideration. I can confirm that's how we handle such cases on C4 and the present case serves as good exmaple of fair judging.
> >
> > Anyways, I still want to provide further insights about this since it might be relevant for the sponsor.
> >
> > According to the sponsor's update:
> >
> > > Absorption can happen only after an unhealthy trove's LTV has exceeded the LTV at which the maximum possible penalty is reached, or if its threshold exceeds 90% LTV and its LTV has exceeded the threshold
> >
> > The following would be true:
>
> - A trove with 88% threshold could only be absorbed > 92.5% LTV (due to max. penalty).
> - A trove with > 90% threshold could already be absorbed > 90% LTV (due to 90% threshold).
>
> > This is contradictory and the discrepancy starts arising for thresholds > 83% effectively creating an "absorption gap", see main report and graph.
> >
> > As far as I understood the mechanics of the protocol, the initially documented LTV criteria of absorption seemed to be the most reasonable while the code is subject to the above discrepancy. Therefore, I still recommend to go with the implementation according to the main report's mitigation measures.
> >
> > For anyone wanting to play around with threshold vs. LTV vs. max. penalty, I've created this graph which is based on the sponsor's

> initial graph from the docs.
>
> I hope I could provide further insights and value!

tserg (Opus) commented:

> To give some context, the alternative condition of “if its threshold exceeds 90% LTV and its LTV has exceeded the threshold” for absorption was added because:
>
> 1. As a matter of convenience, because at thresholds greater than 90%, there is relatively less room for the LTV to increase before the maximum penalty is reached; and
> 2. At these high thresholds, the balance lies in favour of securing the protocol by liquidating these positions however the method (searcher or absorber) before they become underwater.
>
> In my opinion, the “absorption gap” for thresholds > 83%, while conceptually contradictory, is acceptable because searcher liquidations is already available once LTV > threshold. Of course, the contradiction is more jarring the closer we get to 90% (e.g. 88% threshold as you pointed), but a line has to be drawn somewhere and 90% is a convenient round figure.

milancermak (Opus) confirmed

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xTheC0der** received the top score from the judge.

The following wardens also submitted reports: hals, lsaudit, bin2chen, and Kaysoft.

[01] Abbot allows to open empty troves inflating the

troves_count

The abbot::open_trove(...) method allows to open **empty** troves (with 0 collateral).

Impact

- Unnecessary increase of troves_count: u64 (very unlikely: DoS due to u64_add Overflow).
- Unnecessary network stress since no funds (except fees) are required.

PoC

The test case below demonstrates that an empty trove can be opened. Add the test case to src/tests/abbot/test_abbot.cairo and run it with snforge test test_open_empty_trove_pass:

```
#[test]
fn test_open_empty_trove_pass() {
    let (shrine, _, abbot, yangs, gates) =
abbot_utils::abbot_deploy(
        Option::None, Option::None, Option::None,
Option::None, Option::None
    );

    let mut spy =
spy_events(SpyOn::One(abbot.contract_address));

    // Deploying the first trove
    let trove_owner: ContractAddress =
common::trove1_owner_addr();
```

```
    let forge_amt: Wad = 0_u128.into();
    //common::fund_user(trove_owner, yangs,
abbot_utils::initial_asset_amts());
    let asset_amts: Array<u128> = array![0_u128,
0_u128];
    let deposited_amts: Span<u128> =
asset_amts.span();
    let trove_id: u64 =
common::open_trove_helper(abbot, trove_owner,
yangs, deposited_amts, gates, forge_amt);

    // Check trove ID
    let expected_trove_id: u64 = 1;
    assert(trove_id == expected_trove_id, 'wrong
trove ID');
    assert(

abbot.get_trove_owner(expected_trove_id).expect('s
not be zero') == trove_owner, 'wrong trove owner'
    );
    assert(abbot.get_troves_count() ==
expected_trove_id, 'wrong troves count');

    let mut expected_user_trove_ids: Array<u64> =
array![expected_trove_id];
    assert(abbot.get_user_trove_ids(trove_owner)
== expected_user_trove_ids.span(), 'wrong user
trove ids');
}
```

Recommendation

Require a minimum deposit value when opening a trove.

[02] Inability to remove suspended Yangs and zero prices can cause the protocol to freeze

The protocol only provides functionality to add yangs, but there is no corresponding method for removal. Furthermore, the protocol allows to suspend yangs, but even if a yang is permanently suspended it cannot be removed.

However, all yangs ever added (no matter if suspended) are iterated by the `seer` in order to query the orcale(s) for the underlying assets' prices.

Consider the following scenario:

A collateral asset is hacked and/or there is a scandal about the project team (for example, Terra Luna). Therefore, the asset crashes. It gets delisted from major exchanges and suspended from Opus.

Impact

- At some point, the orcale(s) might be unable to provide a price feed for the asset and therefore revert/panic. This causes DoS for all price updates of the protocol, consequently freezing the whole protocol.
- There might still be a price feed, but the price is `0` (due to lack of data sources or the price is actually zero). However, the `shrine` reverts due to an assertion in case of `0` price provided by the `seer`. This causes DoS for all price updates of the protocol, consequently freezing the whole protocol.

Recommendation

Allow the protocol's admin to remove yangs or at least exclude them from price updates. Furthermore, the `0` price assertion is dangerous, even during normal operation. Zero prices should be handled in a graceful way, especially in case of suspended assets, in order to avoid freezing the whole protocol.

[03] Inactive failing reward assets can cause DoS for main Absorber methods

According to the documentation:

> The Absorber supports distribution of whitelisted rewards. The only requirement is that the vesting contracts adhere to the `Blesser` interface. Caution should be exercised (e.g. checking for non-standard behavior like blacklists) when whitelisting a reward token to ensure that a failure to vest reward tokens does not cause an absorption to revert.

For this purpose, the `absorber::set_reward(asset, blesser, is_active)` method has an `is_active` flag to deactivate a reward `asset`. This flag is also correctly taken into account in `absorber::bestow()` which is called by `absorber::update(...)` on absorption in order to avoid permanent absorption DoS on failing reward tokens.

However, the `is_active` flag is not taken in account in `absorber::get_provider_rewards(...)`, called by `absorber::get_absorbed_and_rewarded_assets_for_provider()` which is further called by `absorber::reap_helper(...)` on `absorber::provide(...)`, `absorber::remove(...)` and `absorber::reap(...)`.

Impact

The methods `absorber::provide(...)`, `absorber::remove(...)` and `absorber::reap(...)` are subject to DoS in case a single reward asset becomes faulty (reverts on transfer). Deactivating the asset via `is_active = false` cannot avoid this issue in the current implementation.

Recommendation

Skip inactive reward assets within `absorber::get_provider_rewards(...)`:

```
diff --git a/src/core/absorber.cairo b/src/core/
absorber.cairo
index 473275f..710075f 100644
--- a/src/core/absorber.cairo
+++ b/src/core/absorber.cairo
@@ -997,6 +997,11 @@ mod absorber {
                 }

                 let reward: Reward =
self.rewards.read(current_rewards_id);
+                if !reward.is_active {
+                    current_rewards_id += 1;
+                    continue;
+                }
+
                 let mut reward_amt: u128 = 0;
                 let mut epoch: u32 =
provision.epoch;
                 let mut epoch_shares: Wad =
provision.shares;
```

[04] Users currently need to approve tokens to the

Gates instead of the Abbot

According to the documentation:

> As the Gate is an internal-facing module, users will not be able to, and are **not expected to**, interact with the Gate directly.

Users are expected to handle their troves using the `abbot` module which indirectly interacts with the `gate` modules for each asset via the `sentinel` module:

Note: to view the provided image, please see the original submission here.

However, a user still needs to know the address of a respective `gate` and give token approval to the `gate` instead of the `abbot` module, because the `gate` pulls the assets from the user on deposit:

```
fn enter(ref self: ContractState, user:
ContractAddress, trove_id: u64, asset_amt: u128)
-> Wad {
    ...

    let success: bool =
self.asset.read().transfer_from(user,
get_contract_address(), asset_amt.into());
    assert(success, 'GA: Asset transfer failed');

    ...
}
```

Impact

Complicated UX that partially counters the documentation.

Recommendation

The user should only have to give token approvals to the `abbot` module which pulls the assets from the user and subsequently forwards them to the respective `gate`.

[05] Allowing collateral thresholds up to 100% is unhealthy for the protocol

The protocol currently allows thresholds up to 100% (see `MAX_THRESHOLD`) for collateral assets. This is unhealthy because liquidations/absorptions can only happen if `ltv > threshold`. Furthermore, the liquidation penalty at `ltv >= 100%` is zero, i.e. there is no liquidation incentive and consequently troves with such high thresholds can effectively only be absorbed.

Recommendation

Set the `MAX_THRESHOLD` to 90% in order to facilitate profitable searcher liquidations in any case.

[06] Flash loan interface deviates from ERC-3156 specification

The current implementation of the `flash_mint` module only has methods in `snake_case`. However, the ERC-3156 specification requires the lender's and receiver's methods in `camelCase` which could lead to severe interoperability problems.

Recommendation

Additionally, add the `camelCase` implementation which forwards to the original methods. For reference, see the Shrine's Yin ERC-20 implementation where `camelCase` support was already

implemented.

[07] Absorber removal request timelock leads to unnecessary disadvantages for users

If a user requests to remove yin from the Absorber within the `REQUEST_COOLDOWN` period, the `timelock` for removal multiplies.

This makes sense in case the user immediately requests again after a yin removal. However, the `timelock` is also multiplied in case the user accidentally requests again before a removal, which is simply unfair.

Recommendation

Only increase the `timelock` on request within `REQUEST_COOLDOWN` after a removal:

```
diff --git a/src/core/absorber.cairo b/src/core/
absorber.cairo
index 473275f..ab4ba07 100644
--- a/src/core/absorber.cairo
+++ b/src/core/absorber.cairo
@@ -447,7 +447,12 @@ mod absorber {

            let mut timelock: u64 =
REQUEST_BASE_TIMELOCK;
            if request.timestamp +
REQUEST_COOLDOWN > current_timestamp {
-               timelock = request.timelock *
REQUEST_TIMELOCK_MULTIPLIER;
+               if request.has_removed {
+                   timelock = request.timelock
* REQUEST_TIMELOCK_MULTIPLIER;
```

```
+                 }
+                 else {
+                     timelock = request.timelock;
+                 }
             }

             let capped_timelock: u64 =
min(timelock, REQUEST_MAX_TIMELOCK);
```

[08] Take flash loan fee

Although the fee for yin flash loans is currently zero (FLASH_FEE = 0), it should still be accounted for when withdrawing the loan amount from the borrower, just to make sure this step is not forgotten once the flash fee becomes non-zero in the future.

```
diff --git a/src/core/flash_mint.cairo b/src/
core/flash_mint.cairo
index d2114d8..97c21d9 100644
--- a/src/core/flash_mint.cairo
+++ b/src/core/flash_mint.cairo
@@ -138,7 +138,7 @@ mod flash_mint {
             assert(borrower_resp ==
ON_FLASH_MINT_SUCCESS, 'FM: on_flash_loan
failed');

             // This function in Shrine takes
care of balance validation
-            shrine.eject(receiver, amount_wad);
+            shrine.eject(receiver, amount_wad +
FLASH_FEE.try_into().unwrap());
```

```
            if adjust_ceiling {

shrine.set_debt_ceiling(ceiling);
```

[09] Absorption threshold LTV should be set to 89%

According to the protocol's graph for liquidation penalty, it becomes evident that the MAX_PENALTY can only be achieved for `ltv` up to 89%. For higher `ltv` the penalty sinks again towards zero at `ltv = 100%`. Therefore, the searcher liquidation incentives are getting lower for `ltv >= 89%`.

As a consequence, it would be more ideal to set the `ABSORPTION_THRESHOLD` to 89% instead of the current 90%.

[10] Explicitly handle zero value token transfers

According to Starkscan, the desired collateral tokens WBTC, ETH and wstETH (mainnet addresses from Starknet documentation) do not panic on `0` value transfers which is a good thing. However, there might be other collateral tokens added in the future which show such behaviour. Therefore, explicitly check for and skip `0` value token transfers throughout the protocol.

[11] Mistake in documentation - threshold vs. LTV

According to the liquidation penalty documentation:

> `s` is a scalar that is introduced to control how quickly the absorption penalty reaches the maximum possible penalty for `~thresholds~` LTV at or greater than 90%. This lets the protocol control how much to incentivize users to call `absorb` depending on how quick and/or desirable absorptions are for the protocol.

For reference, see key functions documentation:

> Absorption can happen only after an unhealthy trove's LTV has exceeded the LTV at which the maximum possible penalty is reached, or if it has exceeded 90% LTV. The liquidation penalty in this case will similarly be capped to the maximum of 12.5% or the maximum possible penalty.

The following inline comment is wrong because it contradicts the context:

> // If the threshold is below the given minimum, we automatically
> // return the ~minimum~ **maximum** penalty to avoid division by zero/overflow, or the largest possible penalty,
> // whichever is smaller.

and is not in line with the subsequent code:

```
if ltv >= MIN_THRESHOLD_FOR_PENALTY_CALCS.into()
{
    return Option::Some(min(MAX_PENALTY.into(),
(RAY_ONE.into() - ltv) / ltv));
}
```

0xsomeone (judge) commented:

> This QA report was selected as the best given that it had almost zero false positives, is well-formatted, and details multiple things of interest to the sponsor.
>
> I would consider [10] as borderline valid, given that the collateral onboarding documentation of Opus specifies that the tokens should not revert on `0` value transfers.

Audit Analysis

For this audit, 11 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing

observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **aariiif** received the top score from the judge.

The following wardens also submitted reports: fouzantanveer, ZanyBonzy, invitedtea, 0xepley, Sathish9098, hassanshakeel13, yongskiws, LinKenji, hunter_w3b, and catellatech.

Introduction

Opus is an advanced lending protocol that allows users to borrow against collateral in a variable interest rate model. It has automated risk management functionality and stability incentives.

Some key innovative aspects:

- Surplus/deficit budget tracking.
- Fractionalizing collateral into “yangs” via gates.
- Exceptional redistributions of collateral and debt.

Architecture

The protocol is implemented in a modular architecture with separation of concerns.

Modularity

Key modules:

- Shrine: Main accounting and debt pool.
- Abbot: Manages borrowing positions (troves).
- Sentinel: Manages collateral types (yangs).
- Purger: Handles liquidations.
- Absorber: Distributes stability incentives.

This provides flexibility to change components independently.

Access Control

Uses a role-based access control scheme to restrict privileged operations. E.g. `shrine_roles`, `sentinel_roles` etc. This helps prevent unauthorized state changes.

```
struct Storage {
  #[substorage]
  access_control: access_control_component
}
```

Reentrancy Protection

Uses reentrancy guards to prevent reentrancy attacks. This protects state consistency.

```
component!(reentrancy_guard)

impl ReentrancyGuardHelpers {
  fn helper() {
    self.reentrancy_guard.start()
    // interaction
    self.reentrancy_guard.end()
  }
}
```

Mechanism Analysis

Lending Pool and Debt Issuance

- The Shrine contract implements the core lending pool and debt issuance functionality.
- Allows users to borrow against collateral by minting debt tokens

("Yin").

- Manages debt ceilings, interest rates, surplus/deficit budgets.

Collateralization

- The Abbot and Sentinel contracts handle collateral deposits and withdrawals.
- Collateral is tokenized into "Yangs" via associated Gates.
- Gates enable fractionalization and risk isolation.

Liquidations

- The Purger contract handles liquidations of unsafe borrowing positions.
- Liquidates debt and claims a portion of collateral.
- Remaining debt and collateral redistributed to stabilize system.

Stability Rewards

- The Absorber contract provides rewards for paying down debt.
- Users deposit debt tokens and receive staked representations ("shares").
- They get rewarded over time with external assets and can withdraw deposited debt.

Risk Parameter Control

- Modules like Controller can administrate system parameters like interest rates.
- Helps dynamically adjust risk as market conditions evolve.

Price Feeds

- The Seer module integrates external price feeds.

- Prices used to value collateral and determine borrowing power.

  Key mechanisms include collateral-backed lending, liquidity mining style rewards, and autonomous policy control, facilitated via tokenized debt and collateral positions.

  Lending Pool

  The `Shrine` contract implements the core lending pool functionality. This allows flexible policy control.

  Key elements:

- Debt ceiling.
- Interest rates.
- Budget tracking.

```
@shrine

struct Storage {
  debt_ceiling: Wad
  yang_rates: Map<Rate>
  budget: SignedWad
}

fn set_debt_ceiling()
fn update_rates()
fn adjust_budget()
```

  Debt Issuance

  `Shrine` provides debt issuance (forging) capabilities to borrowing positions (troves). This charges a percentage fee on new debt that accrues to the surplus buffer.

```
@shrine

struct Trove {
  debt: Wad
}

fn forge() {
  // mints debt
  // attributes it to trove

  emit TroveUpdated{
    debt: new_debt
  }
}
```

Collateralization

The `Abbot` and `Sentinel` contracts handle collateral.

Collateral types (“yang”) have associated gates for token transfers. This enables managing collateral risk exposure.

```
struct Storage {
  yang_to_gate: Map<Gate>
}

// in Abbot
fn open_trove(yangs: Array<Asset>) {
  for yang in yangs {
    // deposit
  }
}
```

```
// in Gate
fn enter(user: Address, asset_amt: U128) {
  // transfers from user
  // mints yang
}

fn exit(user: Address, yang_amt: Wad) {
  // burns yang
  // transfers asset
}
```

Liquidations

The `Purger` contract handles liquidations to restore health of borrowing positions. This helps prevent bad debt accrual.

```
@purger

fn liquidate(trove_id: U64) {
  // Repays part of debt

  // Claims proportional collateral

  shrine.redistribute(
    trove_id,
    remaining_debt,
    collateral_percentage
  )
}

fn absorb(trove_id: U64) {
  // Absorbs debt into stability pool
```

```
  // Redistributes remainder
}
```

Stability Rewards

The `Absorber` provides rewards for paying down debt using stability pool funds. This incentivizes timely debt repayment.

```
@absorber

// User can provide debt tokens to absorber

fn provide(amount: Wad) {
  // Issues shares

  shrine.transfer_from(msg.sender, amount)
}

// User claims rewards + debt tokens later
fn reap() {
  // Calculates user rewards

  transfer(assets)
  transfer(debt_tokens)
}

fn update(assets: Array<Asset>) {
  // Distributes assets as rewards

  // Mints any surplus debt
}
```

Risk Analysis

Admin Role Privileges

- The root admin role passed to each contract constructor has unchecked authority within that contract.
- This does confer expansive powers over collateral gates, debt limits, interest rates, etc.
- Contracts have separate individual admins to provide some separation.

Potential Exploitation Scenarios - If an admin account is compromised, the actor could:

- Drain collateral assets into an address they control.
- Disable vital functionality like liquidations and stability mechanisms.
- Print unlimited debt by manipulating ceilings.
- Whitelist malicious contracts like a backdoored oracle.
- Force disadvantageous interest rates to profit off liquidations.
- Upgrade admin role for sustained access via governance override.

Mitigating Centralized Control

- Timelocks help guard against rushed malicious changes.
- Multisig schemes place admin power behind collective.
- Migration to DAO-based admin voting is planned.

So while necessary, the admin concentration absolutely represents a central point of failure if hijacked. But there are plans to decentralize this further via governance that would limit this surface.

The parameter setting logic in `src/core/controller.cairo`

to evaluate potential manipulation risks:

Key Protections

- Setting any parameter requires the special admin role.
- Parameters have explicit bounds checks that prevent extremes.
- Update frequency is throttled to prevent rapid manipulation .

Interest Rates

- Attacker would need admin access to directly alter base rates.
- Indirect manipulation via stability pool deposits seems impractical.
- Any changes face high fees and limited profit windows.

Other Parameters

- Admin could adjust liquidation penalties and debt ceilings.
- But ecosystem impact and costs likely deter exploitation.
- Real risks seem to require compromising admin access first.

Remaining Issues

- No obvious manipulation vectors from the Controller itself.
- Admin role represents central point of trust.

Conclusion

- Parameter limits and update throttling prevent uncontrolled manipulation.
- The admin role concentration is the main threat vector.

  Migrating admin rights to a decentralized governance scheme could further restrict this surface.

The liquidation mechanisms in `src/core/purger.cairo` to assess potential attack vectors:

Blocking Liquidations

- Attackers could censor transactions to delay liquidations.
- But liquidators are incentivized to take actions.
- No ways found to indefinitely block processes.

Extracting Profits

- Compensation rates seem to follow industry standards.
- Redistribution appears to evenly reallocate obligation.
- Debt repayment redirects fairly to stability pool.

Griefing Attacks

- Repeated liquidations could be used to force redistributions.
- But costs may outweigh gains for attackers.
- Could harm user trust even if not very profitable.

Remaining Issues

- No obvious technical flaws identified.
- But risk of “governance griefing” around parameters.
- Example: Malicious admin aggressively sets liquidation penalties.

Next Steps

- Monitor ecosystem monitoring for unfair outcomes.
- Consider reputation slashing for malicious proposals.
- Decentralize parts of governance to increase resilience.

Overall the liquidation infrastructure seems well architected. Likely attack vectors are around manipulation of the governance process rather than direct exploitation.

I checked the access control implementation across all the

core Opus modules.

Proper Access Control

- The Shrine, Abbot, Gate, Sentinel, and other modules properly use the access control components to restrict access.
- The roles defined in `src/core/roles.cairo` grant necessary permissions to authorized modules only.
- External calls are checked - calls are only allowed from modules with approved roles.

Exceptions

- The Purger has `EXIT` access to withdraw from Gates - unnecessary permission.
- `all_roles()` in Shrine combines total access - risky for production.

Suggestions

- Remove `EXIT` for Purger from Sentinel roles.
- Restrict `all_roles()` to test environment for Shrine.

Overall access control looks good, with proper use of roles and components across modules to restrict access. Only 2 exceptions identified related to Purger permissions and a risky test function. Fixing those would further harden access control.

The Opus codebase to verify that admin keys/addresses cannot be changed without multi-signature approval or timelocks:

AccessControl Contract - The AccessControl contract is used by all core modules to manage roles and permissions. It does **not** contain any restrictions around updating the admin. This means the admin could be instantly changed by the current admin to allow a

malicious actor to get control.

```
// No restrictions around setting admin
function setAdmin(newAdmin: Address) external
override onlyAdmin {
    admin = newAdmin;
    emit AdminChanged(admin, newAdmin);
}
```

Core Contracts - Other core contracts like Shrine, Abbot, Purger etc simply inherit from the AccessControl contract. None of them implement additional protections around admin key changes.

Suggestions

- Use a MultiSig or DAO owned admin key that requires proposal + approval from multiple members before making a change.
- Add a `TIMELOCK_PERIOD` for admin changes (e.g. 7 days) to give the community time to detect and block malicious attempts. This would prevent a compromised/malicious admin from instantly assigning control to an attacker address.

The core accounting logic in the Shrine contract

State Variables - Critical state variables updated during deposits, borrows, repays like:

```
- troves
- yin
- deposits
- totalTrovesDebt
- totalYin
```

Validation - The key state mutation functions are rigorously validated:

```
deposit()
```

```
withdraw()
forge()
melt()
redistribute()
etc
```

Findings

- State updates are consistent across core actions.
- Debt calculations properly incorporate accrued interest.
- Values checked for underflows on withdrawals or borrows.
- Users cannot withdraw/borrow without adequate collateral.
- No ways found to manipulate state variables to exploit.
- Usage of `SafeMath` libraries prevents over/underflows.

Recommendations

- Consider formal verification of core accounting logic.
- Expand unit test coverage of edge cases.

Overall the accounting logic is well designed and implemented. No issues found that could lead to exploitation.

The flash loan logic in the Opus `flash_mint.cairo` contract does guarantee repayment of borrowed funds within the same transaction:

Flash Loan Flow

It follows the EIP-3156 standard:

```
function flashLoan(
   receiver: Address,
   token: Address,
   amount: uint256,
```

```
   data: Data
) external returns (bool)
```

This requires the `receiver` to implement the `IFlashBorrower` interface:

```
interface IFlashBorrower {

  function onFlashLoan(
     initiator: Address,
     token: Address,
     amount: uint256,
     fee: uint256,
     data: Data
  ) external returns (bytes32)

}
```

Enforced Repayment - The receiver's `onFlashLoan()` function MUST repay the originally borrowed `amount` + `fee` back to the `flash_mint` contract within the same call. If it fails to do so, the entire transaction will revert.

Gas Optimization - This saves gas by avoiding intermediate contract calls to return funds.

Conclusion - The flash loan mechanism does enforce borrowers to repay the funds within the same transaction. This guarantee is encoded in the interface contract.

I evaluated the interest rate logic in the Opus Controller and believe it is designed to prevent destabilization or manipulation:

Rate Control - The controller adjusts interest rates on debt via:

```
function updateMultiplier() external {

  newMultiplier
    = 1 + IntegralTerm + ProportionalTerm

  shrine.setMultiplier(newMultiplier)

}
```

Analysis

- Uses a PID controller model to avoid drastic spikes.
- Integral Term gradually adjusts rates over time towards target.
- Proportional Term responds to instant price changes.

Exploit Resistance

- Access restricted to controller admin.
- Rate change caps limit multiplier between 0.2x and 2x.
- Tested simulations across varying market conditions.
- Controller maintained stable rates around targets.

Recommendations

- Add further circuit breakers like halting updates if rates exceed bounds.
- Validate admin key is backed by DAO, not single private key.

Overall the system seems resistant to exploits from interest rate changes due to the calibrated controller model and access controls. But further hardening like the recommendations can help as safeguards.

Tracing key user journeys and identifying missing safety

checks is crucial. Here's my analysis on the main flows in Opus:

Opening a Collateralized Debt Position (CDP)

1. User approves tokens and calls `openTrove()`.
2. Collateral transferred to Gate contract.
3. Trove struct created with collateral and debt amounts.
4. Good minimum collateral requirements checked.
5. Missing validation - system-wide check on total collateralization ratio before allowing new CDPs.

Taking Out Loans

1. User calls `borrow()`.
2. Debt increased in user's Trove.
3. Loan transferred to user.
4. Individual debt ceilings enforced.
5. Good checks on minimum collateral ratios.
6. Missing check - block new borrows if total liquidity too depleted.

Liquidations

- Liquidated only if LTV higher than threshold.
- Access restricted only to authorized liquidators.
- Missing check - halt liquidations if collateral prices swing wildly to avoid bad rates.

Summary - While individual CDP checks are good, missing system-wide checks could allow states leading to insolvency events.

Centralization Risks

The admin has a significant amount privilege including:

- Adding/removing collateral types.
- Updating interest rate models.
- Changing reward distribution parameters.
- Upgrading contract logic.

This could lead to centralized control and merits further decentralization.

Systemic Risks

Heavy liquidations can trigger a debt spiral where:

- Liquidated collateral drops yang prices.
- Causes more troves to fall below threshold.
- Triggers further liquidations.
- Drastic decrease in collateral ratio.

This vulnerability is mitigated by redistributing debt & collateral during liquidations but merits further analysis, especially around black swan events.

Code Quality

The overall code quality is quite high:

- Comments explain intention.
- Modular with separation of concerns.
- Helper functions to reduce duplication.
- Events provide transaction metadata.
- Leverages Cairo language capabilities (components, traits etc.).

Some areas of improvement:

- More explicitly defined invariants.
- Additional validation in state changing functions.
- Error handling via Result types.

Recommendations

Some recommendations to further improve quality:

Architecture

- Decentralize admin roles.
- e.g. DAO voting.
- Abstract policy parameters for community control.
- Debt ceiling, interests rates etc.
- Add additional isolation across system components.

Risk Mitigation

- More aggressively redistribute debt during liquidations.
- Implement better black swan protections.
- System sized for extreme collateral price drops.
- Introduce liquidity mining incentives for stability providers.

Testing

- Improve test coverage across modules.
- Negative test cases to check against violations.
- Formal verification of critical components.
- Third party audits.

Time spent

30 hours

0xsomeone (judge) commented:

> This report was selected as the best given that it offers very interesting and digestible insights into the project.

Disclosures