Fluid Protocol

Security Assessment

October 20th, 2024 — Prepared by OtterSec

Alpha Toure shxdow@osec.io

Renato Eugenio Maria Marziano renato@osec.io

James Wang james.wang@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Hydrogen Labs engaged OtterSec to assess the `fluid-protocol` program. This assessment was conducted between August 20th and October 14th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 21 findings throughout this audit engagement.

In particular, we identified a vulnerability concerning the staleness check in the oracle price retrieval function, which may result in constant reverts if there is a time drift between Fuel's local timestamp and Pyth's published time (OS-FLP-ADV-04). We also found another issue where the reward snapshot is not updated for partially liquidated borrowers, resulting in a misalignment between the recorded rewards and the actual collateral/debt after liquidation (OS-FLP-ADV-00). Additionally, upon cancellation of a partial redemption due to insufficient debt, the stake and reinsertion in the sorted list are not properly updated (OS-FLP-ADV-01).

Furthermore, there are several functional inconsistencies, including a lack of a check for zero amount transfers (OS-FLP-ADV-12) and the processing of excessive USDF refunds without verifying the message asset ID when the debt is zero (OS-FLP-ADV-11). We also highlighted flaws in the u128 implementation (OS-FLP-ADV-13).

We also made recommendations to ensure adherence to coding best practices (OS-FLP-SUG-06) and suggested the removal of unutilized and redundant code within the system for increased readability (OS-FLP-SUG-04, OS-FLP-SUG-05). We further advised incorporating additional checks within the codebase for improved robustness and security (OS-FLP-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at ttps://github.com/Hydrogen-Labs/fluid-protocol. This audit was performed against commit 54cac4d.

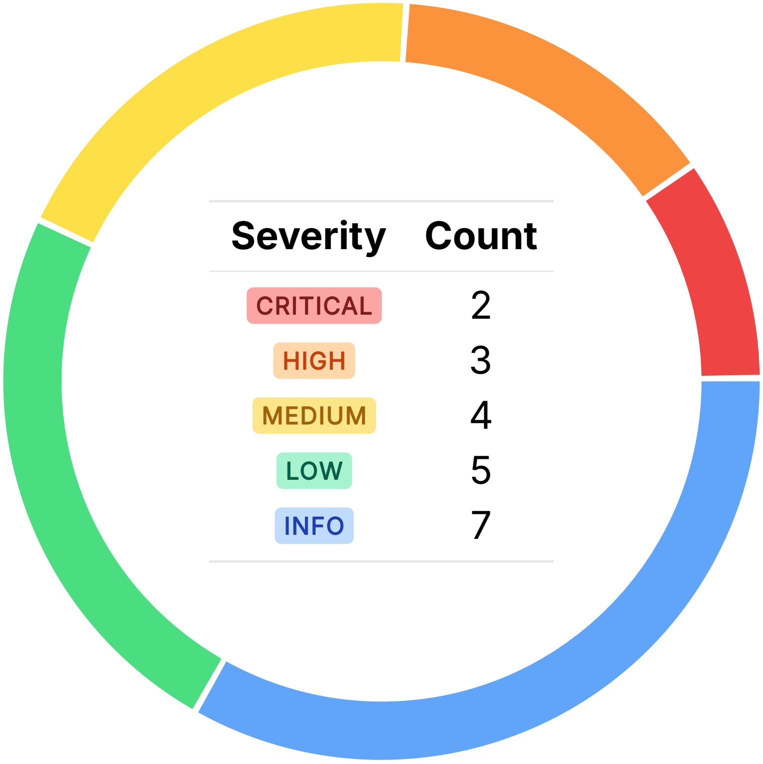
A brief description of the program is as follows:

Name	Description
fluid-protocol	A decentralized protocol that allows holders of certain assets to obtain maximum liquidity against their collateral without paying interest.

03 — Findings

Overall, we reported 21 findings.

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



Severity	Count
CRITICAL	2
HIGH	3
MEDIUM	4
LOW	5
INFO	7

04 — Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-FLP-ADV-00	CRITICAL	RESOLVED ✓	The reward snapshot is not updated for partially liquidated borrowers, resulting in a misalignment between the recorded rewards and the actual collateral/debt after liquidation.
OS-FLP-ADV-01	CRITICAL	RESOLVED ✓	Upon cancellation of a partial redemption due to insufficient debt, the stake and reinsertion in the sorted list are not updated to reflect the changes made during the previous `apply_pending_rewards` operation.
OS-FLP-ADV-02	HIGH	RESOLVED ✓	The current implementation in `oracle_interface` utilizes an incorrect structure for retrieving Pyth oracle prices.
OS-FLP-ADV-03	HIGH	RESOLVED ✓	`require_no_undercollateralized_troves` incorrectly utilizes `get_first` to retrieve the first trove, which is sorted by NICR from high to low.
OS-FLP-ADV-04	HIGH	RESOLVED ✓	The current staleness check in `get_price` may result in constant reverts if there is a time drift between Fuel's local timestamp and Pyth's `publish_time`.

OS-FLP-ADV-05	MEDIUM	RESOLVED ✓	The call to `apply_pending_rewards` during redemption may alter the order of troves, disrupting the expected sequence of redemptions that prioritizes troves with the lowest Inverse Collateral Ratio (ICR).
OS-FLP-ADV-06	MEDIUM	RESOLVED ✓	`batch_liquidate_troves` in trove-manager is missing checks for active troves, sorted order, and duplicate entries.
OS-FLP-ADV-07	MEDIUM	RESOLVED ✓	`internal_redistribute_debt_and_coll` utilizes `total_stakes` for reward distribution, which fails to account for pending rewards, resulting in an inaccurate calculation of asset and debt redistribution among troves.
OS-FLP-ADV-08	MEDIUM	RESOLVED ✓	In `internal_find_insert_position`, the condition incorrectly references `prev_id` instead of `next_id`.
OS-FLP-ADV-09	LOW	RESOLVED ✓	`renounce_admin` and `owner` fail in the `ProtocolManager` because Sway-libs does not utilize `storage.owner`.
OS-FLP-ADV-10	LOW	RESOLVED ✓	`internal_is_empty` may inaccurately return false when the asset's size is not found, potentially resulting in unnecessary gas consumption in `internal_valid_insert_position`.
OS-FLP-ADV-11	LOW	RESOLVED ✓	In `close_trove`, excessive USDF refunds are processed without verifying the message asset ID if the debt is zero.

OS-FLP-ADV-12	LOW	RESOLVED ✓	`send_asset` lacks a check for zero transfer amounts.
OS-FLP-ADV-13	LOW	RESOLVED ✓	The `fluid_math` library contains flawed implementations for the `U128` type, particularly in `is_power_of_two` and `modulo`.

Reward Snapshot Misalignment CRITICAL OS-FLP-ADV-00

Description

In `internal_apply_liquidation` within trove-manager, when a `trove` is partially liquidated, it is important to update the borrower's reward snapshot (the record of pending collateral and debt rewards). This is because the liquidation process calculates and accounts for the rewards that have accrued up to the moment of liquidation.

>_ *trove-manager-contract/src/main.sw* SWAY

```sway
fn internal_apply_liquidation(
    borrower: Identity,
    liquidation_values: LiquidationValues,
    upper_partial_hint: Identity,
    lower_partial_hint: Identity,
) {
    let asset_contract_cache = storage.asset_contract.read();
    // partial liquidation reinserted into sorted troves
    if (liquidation_values.is_partial_liquidation) {
        let mut trove = storage.troves.get(borrower).read();
        [...]
    }
    [...]
}
```

The snapshot stores values representing the borrower's position at a particular point in time. After liquidation, the borrower's new `trove` position (with updated collateral and debt) should reflect a fresh starting point for accumulating future rewards. Failing to update the snapshot after a partial liquidation may result in double-counting rewards or incorrect reward calculations in the future because the borrower's old snapshot would still reflect the pre-liquidation state.

Additionally, in `internal_get_totals_from_batch_liquidate`, `usdf_in_stability_pool` is utilized, and as a result, the function will always refer to the initial amount of `USDF`, ignoring the fact that some `USDF` has already been utilized.

Remediation

Update the reward snapshot for the borrower immediately after the partial liquidation in `internal_apply_liquidation`, and utilize `vars.remaining_usdf_in_stability_pool` during liquidation calculations in `internal_get_totals_from_batch_liquidate`.

Patch

Resolved in a9279b2.

Failure to Update State on Redemption Cancellation CRITICAL OS-FLP-ADV-01

Description

`internal_redeem_collateral_from_trove` fails to update the stakes and reinsert the trove into the sorted list when the redemption is canceled. The cancellation condition occurs when the new debt after redemption falls below a predefined minimum debt limit (`MIN_NET_DEBT`). In such cases, the function sets the `cancelled_partial` flag to true and returns early without applying the redemption changes.

>_ *trove-manager-contract/src/main.sw* SWAY

```
#[storage(read, write)]
fn internal_redeem_collateral_from_trove(
   [...]
) -> SingleRedemptionValues {
    [...]
    else {
        // Calculate the new nominal collateralization ratio
        let new_nicr = fm_compute_nominal_cr(new_coll, new_debt);

        // If the new debt is below the minimum allowed, cancel the partial redemption
        if (new_debt < MIN_NET_DEBT) {
            single_redemption_values.cancelled_partial = true;
            return single_redemption_values;
        }
        [...]
    }
    [...]
}
```

Since these values would have been altered during `apply_pending_rewards` , failing to update the stakes and reinsert the trove into the sorted list will result in inconsistencies in the state of the contract.

Remediation

Track a compounded value as stake instead of collateral directly. This eliminates the need to update stake when distributing rewards. Also consider pending rewards in NICR calculation to remove the need for reordering troves on reward distribution.

Patch

Resolved in f8006a0 and 89ba7f0.

Utilization of Incorrect Pyth Price Structure HIGH

OS-FLP-ADV-02

Description

The current implementation utilizes an incorrect method to retrieve prices from the Pyth oracle. The `PythPriceFeedId` and `PythPrice` in the current implementation are incorrect representations of how Pyth handles price feeds. Additionally, the `PythPrice` structure in the current implementation lacks critical information, such as the confidence interval and the `exponent`. These missing fields are essential for accurately interpreting the price feed data.

>_ *libraries/src/oracle_interface.sw* SWAY

```sway
// The incorrect PythPrice structure and method.

abi PythCore {
    #[storage(read)]
    fn price(price_feed_id: PythPriceFeedId) -> PythPrice;
    [...]
}

pub struct PythPrice {
    pub price: u64,
    pub publish_time: u64,
}
```

Furthermore, when utilizing price feeds from Pyth, it is considered best practice to check the confidence interval before using the price for any operation. Ignoring the confidence interval may expose the system to risk due to reliance on potentially inaccurate prices.

Remediation

Ensure to utilize the correct Pyth structure, which includes the `confidence` and `exponent` fields, and verify the confidence interval before utilizing the price. Also, refer to the Pyth documentation on best practices for utilizing Pyth price feeds to ensure that the system follows recommended guidelines for safety and reliability.

Patch

Resolved in 995d2b3, 847cd36 and fdd71d1.

Incorrect Trove Retrieval HIGH OS-FLP-ADV-03

Description

In `require_no_undercollateralized_troves`, which checks to ensure that there are no undercollateralized troves in the system, the troves are incorrectly retrieved from the sorted structure. The troves are sorted by their Instantaneous Collateral Ratio (ICR) from high to low. This implies that the trove with the highest ICR (most collateralized) appears first in the sorted list, while the trove with the lowest ICR (potentially undercollateralized) appears last.

>_ *stability-pool-contract/src/main.sw* SWAY

```
fn require_no_undercollateralized_troves() {
    let sorted_troves = abi(SortedTroves, storage.sorted_troves_contract.read().into());
    let mut i = 0;
    while i < storage.valid_assets.len() {
        l[...]
        let price = oracle.get_price();
        let first = sorted_troves.get_first(asset);
        require(
            first == Identity::Address(Address::zero()) || trove_manager
                .get_current_icr(first, price) > MCR,
            "StabilityPool: There are undercollateralized troves",
        );

        i += 1;
    }
}
```

Thus, utilizing `get_first` will give the trove with the highest ICR. However, the current method retrieves the first asset from `sorted_troves`, assuming that this position will be the most at risk for undercollateralization when, in fact, it should be examining the troves with lower ICRs, as they are the ones more likely to be at risk.

Remediation

Replace `get_first` with `get_last` to directly access the trove with the lowest ICR.

Patch

Resolved in 19bb35e.

Possible Flaw in Timestamp Synchronization HIGH OS-FLP-ADV-04

Description

There is a timestamp synchronization inconsistency within `oracle::get_price`. The Pyth oracle's `publish_time` is generated elsewhere, and the Fuel chain's `timestamp()` is based on the local blockchain. Thus, these time sources may not be perfectly synchronized. As a result, `timestamp()` may be behind `publish_time`.

>_ *oracle-contract/src/main.sw* SWAY

```sway
fn get_price() -> u64 {
    // Determine the current timestamp based on debug mode
    let current_time = match DEBUG {
        true => storage.debug_timestamp.read(),
        false => timestamp(),
    };
    // Read the last stored valid price
    let last_price = storage.price.read();
    // Step 1: Query the Pyth oracle (primary source)
    let pyth_price = abi(PythCore, PYTH.bits()).price(PYTH_PRICE_ID);
    // Check if Pyth data is stale
    if current_time - pyth_price.publish_time > TIMEOUT {
        // Step 2: Pyth is stale, query Redstone oracle (fallback source)
    [...]
    }[...]
}
```

`get_price` currently checks if the Pyth data is stale by comparing the difference between `timestamp()` and `publish_time`. It subtracts `publish_time` from `current_time` (Fuel's timestamp). If `publish_time` is ahead of `timestamp()` due to clock drift, the difference may be negative, resulting in the code constantly reverting, even when Pyth's data is valid. This issue is similarly applicable to the Redstone Oracle.

Remediation

Rewrite the staleness check using a safer method that does not assume perfect synchronization, such as employing a saturating subtraction or a `timestamp() > publish_time + TIMEOUT` check instead.

Patch

Resolved in f8006a0.

Trove Redemption Order Violation MEDIUM OS-FLP-ADV-05

Description

In `redeem_collateral` within protocol-manager, there is a potential issue regarding `apply_pending_rewards`. This issue arises from the ordering of troves (collateralized debt positions), which is based on their Nominal Individual Collateral Ratio (NICR). Troves with the lowest NICR are prioritized for redemptions, ensuring that the system remains stable by liquidating or redeeming collateral from riskier positions first.

>_ protocol-manager-contract/src/main.sw SWAY

```
fn redeem_collateral(
    max_itterations: u64,
    partial_redemption_hint: u64,
    upper_partial_hint: Identity,
    lower_partial_hint: Identity,
) {
    [...]
    // Iterate through troves, redeeming collateral until conditions are met
    while (current_borrower != null_identity_address() && remaining_usdf > 0
        ↪  &&remaining_iterations > 0) {
        [...]
        // Apply pending rewards to ensure up-to-date trove state
        trove_manager_contract.apply_pending_rewards(current_borrower);
        [...]
    }
    [...]
}
```

During the redemption process, the protocol is expected to redeem from troves starting with the lowest NICR. However, if `apply_pending_rewards` is called during redemption, the NICR of the current trove may change before the redemption occurs. This change may render the trove safer (higher NICR) or riskier (lower NICR), altering its position relative to other troves. As a result, the redemption may be processed out of order.

Remediation

Include the pending rewards along with trove collateral and debt when calculating NICR. This ensures the system consistently redeems from the troves with the lowest NICR.

Patch

Resolved in 89ba7f0.

Lack of Validation Checks MEDIUM OS-FLP-ADV-06

Description

`batch_liquidate_troves` in trove-manager is missing checks for active troves, sorted order, and duplicate entries, affecting the proper functioning of the liquidation process. Liquidation should only occur on active troves, and sorting is crucial for efficient liquidation. Also, duplicate entries will result in redundant operations, skewed liquidation results, and unexpected behavior, such as attempting to liquidate the same trove multiple times.

>_ *trove-manager-contract/src/main.sw* SWAY

```
fn internal_close_trove(id: Identity, close_status: Status) {
   [...]
    require_more_than_one_trove_in_system(
        trove_owner_array_length,
        asset_contract_cache,
        sorted_troves_contract_cache,
    );
    [...]
}
```

Remediation

Implement checks to ensure that the troves are active, correctly sorted, and to identify duplicate entries.

Patch

Resolved in 2eff674.

Imprecise Reward Distribution Calculation MEDIUM OS-FLP-ADV-07

Description

The issue in `internal_redistribute_debt_and_coll` within trove-manager, where `total_stakes` is utilized as the denominator for calculating `asset_reward_per_unit_staked`, arises because `total_stakes` represents the sum of all the stakes in the system, but it does not account for pending rewards that may not yet be applied to individual troves. This creates a semantic imprecision in the reward distribution.

>_ *libraries/src/oracle_interface.sw* SWAY

```
fn internal_redistribute_debt_and_coll(debt: u64, coll: u64) {
    let asset_contract_cache = storage.asset_contract.read();
    if (debt == 0) {
        return;
    }
    let asset_numerator: U128 = U128::from_u64(coll) * U128::from_u64(DECIMAL_PRECISION) +
        ↪  U128::from_u64(storage.last_asset_error_redistribution.read());
    let usdf_numerator: U128 = U128::from_u64(debt) * U128::from_u64(DECIMAL_PRECISION) +
        ↪  U128::from_u64(storage.last_usdf_error_redistribution.read());
    let asset_reward_per_unit_staked = asset_numerator /
        ↪  U128::from_u64(storage.total_stakes.read());
    [...]
}
```

If pending rewards are not considered, the `asset_reward_per_unit_staked` and `usdf_reward_per_unit_staked` values will be inaccurately distributed. Thus, troves that have not yet applied their rewards may get less than they are owed, while troves with all rewards applied may receive more than their fair share. Additionally, the `total_stakes_snapshot` and `total_collateral_snapshot` variables remain unutilized or unchanged throughout the trove-manager contract and may be removed.

Remediation

Ensure that the denominator in the reward calculation accounts for both the active stakes and the pending rewards.

Patch

Resolved in f8006a0.

Inconsistency in the Validation Check MEDIUM OS-FLP-ADV-08

Description

There is an inconsistency in the condition used to validate `next_id` in `internal_find_insert_position` within the sorted trove. This condition checks whether the `next_id` should remain valid based on two criteria: whether the `next_id` contains a valid entry associated with the specified asset, or whether the NICR of the new entry is less than the nominal ICR of the previous node. However, given the intention of the function, which is to find the correct insertion position based on the NICR, the second part of the condition should actually reference `next_id` instead of `prev_id`.

>_ *sorted-trove-contract/src/main.sw* SWAY

```sway
fn internal_find_insert_position(
    nicr: u64,
    prev_id: Identity,
    next_id: Identity,
    asset: AssetId,
    trove_manager_contract: ContractId,
) -> (Identity, Identity) {
    [...]
    if (next_id != null_identity_address()) {
        if (!internal_contains(next_id, asset)
            || nicr < trove_manager.get_nominal_icr(prev_id))
        {
            next_id = null_identity_address()
        }
    }
    [...]
}
```

Remediation

Utilize `next_id` instead of `prev_id` in the `if` condition.

Patch

Resolved in c75449f.

Improper Ownership Management LOW OS-FLP-ADV-09

Description

In `ProtocolManager`, `renounce_admin` allows the contract admin to renounce their privileges via `storage.owner.write`, while `owner` reads the contract's owner from `storage.owner.read`. However, the sway-libs does not use `storage.owner` to store or manage ownership; thus, these functions will not work as intended.

>_ *protocol-manager-contract/src/main.sw* SWAY

```
#[storage(read, write)]
fn renounce_admin() {
    only_owner();
    storage
        .owner
        .write(State::Initialized(Identity::Address(Address::zero())));
}
#[storage(read)]
fn owner() -> State {
    storage.owner.read()
}
```

Remediation

Align the contract's ownership logic with the way Sway Library currently implements ownership.

Patch

Resolved in d5c54ea and 67142b3.

Discrepancy in Asset Size Check LOW OS-FLP-ADV-10

Description

`internal_is_empty` in sorted-troves-contract checks if there are any troves associated with a given asset. If it fails to find the asset in the `size` map, it currently returns false. In a scenario where an asset is newly added through `add_asset` but has not populated the size map yet, the function will incorrectly indicate that there are troves for that asset when, in fact, there are none.

>_ *sorted-troves-contract/src/main.sw* SWAY

```sway
#[storage(read)]
fn internal_is_empty(asset: AssetId) -> bool {
    match storage.size.get(asset).try_read() {
        Some(size) => return size == 0,
        None => return false,
    }
}
```

When a new asset is introduced via `add_asset`, it may not immediately populate the corresponding entry in the `size` map. Therefore, the absence of an entry for that asset signifies that there are no troves yet associated with it. Returning false when an asset is not found suggests that there are existing troves for that asset, which may mislead other parts of the contract logic, such as the check in `internal_valid_insert_position`, which will fail and result in the wastage of gas while unnecessarily searching for a position.

Remediation

Modify `internal_is_empty` to return true if the asset is not found in the `size` map.

Patch

Resolved in 99b6327.

Missing Asset ID Check for Refunds on Zero Debt LOW OS-FLP-ADV-11

Description

The vulnerability concerns the refund logic for excess USDF in `close_trove` when the debt is zero. If the amount of USDF sent (`msg_amount`) is greater than the debt, the difference is treated as ”excess” and is refunded to the borrower. However, the issue arises from the fact that there is no asset type check when the debt is zero. The `msg_asset_id` check is only done when the debt is greater than zero.

>_ *borrow-operations-contract/src/main.sw* SWAY

```
// Close an existing trove
#[storage(read, write), payable]
fn close_trove(asset_contract: AssetId) {
    [...]
    active_pool.send_asset(borrower, coll, asset_contract);
    if (debt < msg_amount()) {
        let excess_usdf_returned = msg_amount() - debt;
        transfer(borrower, usdf_asset_id, excess_usdf_returned);
    }
    storage.lock_close_trove.write(false);
}
```

This implies that if the debt happens to be zero and the borrower sends any asset type (not necessarily USDF), the excess refund logic may execute. This would allow a malicious borrower to manipulate the system by sending non-USDF tokens or even tokens of no value and receiving USDF in return. While the current system does not seem to have any identified methods to get a debt of exactly zero without closing the trove automatically, it’s always safer to proactively address this, as if this vulnerability were to somehow occur, it could have severe financial consequences.

Remediation

Add a check for `msg_asset_id` even if the debt is zero. This ensures that only valid USDF tokens are involved in the transaction.

Patch

Resolved in ffc5193.

Zero Amount Transfer LOW OS-FLP-ADV-12

Description

Currently, there is a lack of a proper check for a zero amount in `send_asset` in the Active Pool contract. If a caller passes 0 as the amount to be transferred, it may fail the transaction. Also transferring zero amount is nonsensical and should be restricted as it does not serve any purpose.

>_ *active-pool-contract/src/main.sw* SWAY

```
#[storage(read, write)]
fn send_asset(address: Identity, amount: u64, asset_id: AssetId) {
    require_caller_is_bo_or_tm_or_sp_or_pm();
    let new_amount = storage.asset_amount.get(asset_id).read() - amount;
    storage.asset_amount.insert(asset_id, new_amount);
    transfer(address, asset_id, amount);
}
```

Remediation

Add a check to ensure that `amount > 0` before proceeding with the transfer in `send_asset`.

Patch

Resolved in 847cd36.

Faulty U128 Implementation LOW OS-FLP-ADV-13

Description

The `fluid_math` library contains many incorrect implementations for `U128`. It checks if the lower part of the `U128` is a power of two without considering the upper bits. If the upper bits are non-zero, the overall value may still be a power of two, but the current implementation will fail to recognize this. Similarly, `modulo` also neglects the upper bits when calculating.

>_ *libraries/src/fluid_math/numbers.sw* SWAY

```
impl U128 {
    pub fn is_power_of_two(self) -> bool {
        self.lower() != 0 && (self & (self - U128::from(1u64))) == U128::zero()
    }
}

pub fn modulo(self, other: Self) -> Self {
    [...]
    if is_power_of_two(other.lower()) {
        let shift = trailing_zeros(other.lower());
        // upper: self.upper % other.upper,
        // TODO fix this later breaks when upper is 0
        return U128::from(self.lower() & ((1 << shift) - 1));
    }
    [...]
    while dividend >= divisor {
        dividend -= divisor;
    }
    [...]
}
```

This oversight may result in incorrect results when the `U128` value is large enough that its upper bits contribute significantly to the value. However, these functions have not been utilized within the protocol.

Remediation

Given the lack of utilization and the issues identified, it would be prudent to remove these functions.

Patch

Resolved in d5c54ea.

05 — General Findings

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

ID	Description
OS-FLP-SUG-00	The zero-check in `internal_compute_fpt_per_unit_staked` may unnecessarily halt operations when deposits are large, as remainders are already tracked in `last_fpt_error`.
OS-FLP-SUG-01	If no FPT tokens are staked, any USDF or asset fees distributed to the FPT Staking contract will not be allocated to users and will remain stuck in the contract, leading to a loss of those fees.
OS-FLP-SUG-02	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-FLP-SUG-03	`MultiTroveGetter` is currently non-functional due to the absence of implemented logic in `get_multiple_sorted_troves`.
OS-FLP-SUG-04	`internal_adjust_trove` contains multiple redundant checks, which affect the readability and efficiency of the function.
OS-FLP-SUG-05	The codebase contains multiple cases of redundancy that should be removed for better maintainability and clarity.
OS-FLP-SUG-06	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Disruption of Operations due to Zero Check OS-FLP-SUG-00

Description

The `require` statement in `internal_compute_fpt_per_unit_staked` within the stability-pool-contract checks that the calculated `fpt_per_unit_staked` is not zero. The rationale is to prevent a situation where no `FPT` is allocated per unit of `USDF`, which may theoretically happen if the `FPT` issuance is too small compared to the total `USDF` deposited. If this condition occurs, the contract throws an error, disrupting critical contract operations.

>_ *stability-pool-contract/src/main.sw* SWAY

```
fn internal_compute_fpt_per_unit_staked(fpt_issuance: u64, total_usdf_deposits: u64) -> u64 {
    let fpt_numerator = U128::from(fpt_issuance) * U128::from(DECIMAL_PRECISION) +
        ↪  storage.last_fpt_error.read();
    let fpt_per_unit_staked = fpt_numerator / U128::from(total_usdf_deposits);
    require(
        fpt_per_unit_staked != U128::zero(),
        "StabilityPool: compute_fpt_per_unit_staked division rounded to zero error",
    ); // unclear if needed
    [...]
}
```

This situation is highly unlikely because the amount of `FPT` issued should typically be sufficient to cover even a large amount of `USDF` deposits. However, if the total `USDF` deposited becomes disproportionately large compared to the new `FPT` issuance, the division may result in zero. In such a rare case, the contract will throw an error and halt operations such as liquidation, even though the system is capable of functioning normally (since the remainder is handled by `last_fpt_error`).

Remediation

Remove the check to prevent unnecessary contract failures.

Fee Accumulation due to Lack of FPT Staking OS-FLP-SUG-01

Description

The issue in the FPT Staking Contract concerns the handling of fees (in USDF and other asset fees) when no FPT (Fluid Protocol Token) tokens are staked. `increase_f_usdf` and `increase_f_asset` are responsible for distributing collected fees (in USDF and other assets) to stakers of the FPT tokens. However, both functions include a conditional check that verifies whether `total_fpt_staked` is greater than zero. If this condition is false (no FPT tokens are staked), the functions exit without performing any fee distribution.

>_ *fpt-staking-contract/src/main.sw* SWAY

```sway
#[storage(read, write)]
fn increase_f_usdf(usdf_fee_amount: u64) {
    require_is_borrower_operations();
    if (storage.total_fpt_staked.read() > 0) {
        [...]
    }
}

#[storage(read, write)]
fn increase_f_asset(asset_fee_amount: u64, asset_address: AssetId) {
    require_is_protocol_manager(); // we have redeem function in protocol manager, not trove
        ↪ manager inliquity
    if (storage.total_fpt_staked.read() > 0) {
        [...]
        let mut new_f_asset = storage.f_asset.get(asset_address).read() +
            ↪ asset_fee_per_fpt_staked;
        storage.f_asset.insert(asset_address, new_f_asset);
    }
}
```

Consequently, if no FPT tokens are staked at the time fees are collected, the fees will not be allocated to anyone. Thus, if fees are collected but there are no staked FPT tokens, these fees effectively become unclaimed or ”stuck” in the FPT Staking Contract.

Remediation

Implement a mechanism that allows the protocol to reclaim or recycle unclaimed fees, or establish a minimum threshold of FPT tokens that need to be staked to ensure that fees are consistently distributed.

Missing Validation Logic OS-FLP-SUG-02

Description

1. Implement a validation logic to verify that the sum of all vesting schedules adds up to the expected amount to be distributed.
2. Validate whether an asset already exists before attempting to register it again in `register_asset` . While the current implementation does not expose security flaws as it is an admin functionality, adding this check will improve the reliability and maintainability of the protocol.

Remediation

Add the missing validations mentioned above.

Incomplete Query Contract Functionality

OS-FLP-SUG-03

Description

In `MultiTroveGetter`, the `get_multiple_sorted_troves` is currently non-functional because its implementation is commented out and incomplete. `get_multiple_sorted_troves` is meant to retrieve multiple troves from `SortedTroves` but lacks any implemented logic. The body of the function is commented out, and it simply returns an empty vector (`Vec::new`). This implies that the contract, as it stands, does not perform any actions or return any data when invoked.

>_ multi-trove-getter-contract/src/main.sw SWAY

```
impl MultiTroveGetter for Contract {
    // #[storage(read)]
    // fn get_multiple_sorted_troves(
    //     sorted_troves_contract: ContractId,
    //     trove_manager_contract: ContractId,
    //     start_indx: u64,
    //     count: u8,
    // ) -> Vec<CombinedTroveData> {
    //     let mut troves = Vec::new();
    //     let mut index = start_indx;
    //     let mut current_count = 0;

    //     return Vec::new();
    // }
}
```

Remediation

Ensure to implement the whole functionality.

Redundant Checks

OS-FLP-SUG-04

Description

There are multiple redundancies within `BorrowOperations`, specifically in the checks found in `internal_adjust_trove`. The check to ensure that the contract is not paused and the check to verify a valid change (either in collateral or USDF debt) are already present in both `withdraw_usdf` and `internal_adjust_trove`.

>_ *borrow-operations-contract/src/main.sw* SWAY

```
fn internal_adjust_trove(
    [...]
) {
   [...]
    if is_debt_increase {
        require_is_not_paused();
        require_non_zero_debt_change(usdf_change);
    }
    require_non_zero_adjustment(asset_coll_added, coll_withdrawal, usdf_change);
    [...]
    if !is_debt_increase && usdf_change > 0 {
        require_at_least_min_net_debt(vars.debt - vars.net_debt_change);
        require(
            msg_amount() >= vars.net_debt_change,
            "Borrow Operations: caller does not have enough balance to repay debt",
        );
    }
    [...]
}
```

Also, the `require` statement (highlighted above), `msg_amount() >= vars.net_debt_change`, is redundant because if the `msg_amount` is less than `vars.net_debt_change`, the transaction will automatically revert due to insufficient balance. Therefore, explicitly checking this is unnecessary.

Remediation

Remove the redundant checks to improve readability and optimize the code.

Unutilized Code

OS-FLP-SUG-05

Description

1. As the protocol is designed to interact only with explicitly trusted contracts, the risk of reentrancy attacks is significantly reduced. In this case, the protocol controls the entire execution flow, ensuring that it does not inadvertently call untrusted or malicious contracts. Thus, the reentrancy checks within the protocol are redundant and may be removed.
2. In hint-helper-contract, as `get_approx_hint` does not perform any write operations on the contract's storage, it would be appropriate to remove the write attribute.

>_ *hint-helper-contract/src/main.sw* SWAY

```
#[storage(read, write)]
fn get_approx_hint(
    asset: AssetId,
    trove_manager_contract: ContractId,
    cr: u64,
    num_trials: u64,
    input_random_seed: u64,
) -> (Identity, u64, u64);
```

3. There are some duplicate functions in `sorted_troves` that may be removed. `internal_get_first` is identical to `internal_get_head`, and `internal_get_last` is identical to `internal_get_tail`.
4. The `close_status` in `internal_close_trove` does not make sense because it always returns true. As the check does not serve any purpose, it may be removed.

>_ *trove-manager-contract/src/main.sw* SWAY

```
#[storage(read, write)]
fn internal_close_trove(id: Identity, close_status: Status) {
    [...]
    require(
        close_status != Status::NonExistent || close_status != Status::Active,
        "TroveManager: Invalid status",
    );
    [...]
}
```

Remediation

Remove the redundant and unutilized code instances highlighted above.

Code Maturity

OS-FLP-SUG-06

Description

1. `CommunityIssuance` is incorrectly utilizing 31,104,000 seconds as the number of seconds in a year for the `ONE_YEAR_IN_SECONDS` constant, which is approximately five days too short. This miscalculation may result in time-sensitive actions, such as reward distribution or contract transitions, being triggered earlier than intended, leading to less predictable issuance and distribution behaviors.
2. Utilizing `>=` in `require_at_least_min_net_debt` and `require_at_least_mcr` will make the protocol slightly less strict, allowing borrowers to meet the minimum required debt or collateral ratio, instead of exceeding it by a small margin.

>_ *borrow-operations-contract/src/main.sw* SWAY

```
fn require_at_least_min_net_debt(_net_debt: u64) {
    require(
        _net_debt > MIN_NET_DEBT,
        "Borrow Operations: net debt must be greater than 0",
    );
}
fn require_at_least_mcr(icr: u64) {
    require(
        icr > MCR,
        "Borrow Operations: Minimum collateral ratio not met",
    );
}
```

3. In `Oracle`, the **TIMEOUT** constant (the period after which data fetched from the Pyth or Redstone oracles is considered stale) is currently set to four hours. However, Pyth provides real-time price updates with a high frequency, potentially every 400 milliseconds (approximately 0.4 seconds). This implies that Pyth price data may be refreshed multiple times per second. Redstone also provides fresh price data every few minutes. Thus, the TIMEOUT value may result in the utilization of outdated data for hours, even though new and more accurate prices are available.

Remediation

1. Update the constant to the correct number of seconds in a year (31,536,000).
2. Modify the check in `require_at_least_min_net_debt` and `require_at_least_mcr` to utilize `>=` instead of `>`.
3. Reduce the **TIMEOUT** value to a shorter interval so that the system can ensure it uses fresher, more accurate prices from the oracles.

A — Vulnerability Rating Scale

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

CRITICAL

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

Examples:

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

HIGH

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

Examples:

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

MEDIUM

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

Examples:

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

LOW

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

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

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

Examples:

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

B — Procedure

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

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

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

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

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

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

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