Defindex

Security Assessment

March 18th, 2025 — Prepared by OtterSec

Andreas Mantzoutas andreas@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Paltalabs engaged OtterSec to assess the `factory`, `vault`, and `strategies` programs. This assessment was conducted between February 7th and March 10th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 16 findings throughout this audit engagement.

In particular, we identified a critical vulnerability in the bRate calculation that allows attackers to exploit precision loss by manipulating deposits and withdrawals, inflating the bRate, and gaining an unfair profit from the vault's share distribution (OS-DIX-ADV-00). Additionally, when reinvesting, the function sets the minimum expected output to zero during swaps, effectively removing any slippage protection and enabling anyone to sandwich the transaction and extract value from BLND emissions (OS-DIX-ADV-02).

We further highlighted several rounding inconsistencies, including one where the vault suffers from rounding errors when strategies floor minted shares during deposits and ceil burned shares during withdrawals, resulting in small but consistent losses, which may be exploited to grief or extract value benefiting the depositors (OS-DIX-ADV-01), and another in the current division logic while calculating the optimal allocation of amounts for a set of assets, risking underpayment due to rounding down the required amount, which may result in users contributing less than expected to the pool (OS-DIX-ADV-04).

Moreover, it is possible for the first depositor to create a denial-of-service scenario in the vault by setting a reserve to zero in a multi-asset pool, which will result in subsequent deposits failing due to the insufficient managed funds error (OS-DIX-ADV-09). Furthermore, if a user requests to withdraw more than their available balance, the system silently caps the amount without accounting for locked fees (OS-DIX-ADV-07).

We also made recommendations regarding modifications to the codebase for improved efficiency (OS-DIX-SUG-01) and suggested removing multiple instances of redundant code for better maintainability and clarity (OS-DIX-SUG-02). Lastly, we advised including additional safety checks within the codebase to make it more robust and secure (OS-DIX-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/paltalabs/defindex. This audit was performed against commit cde2493.

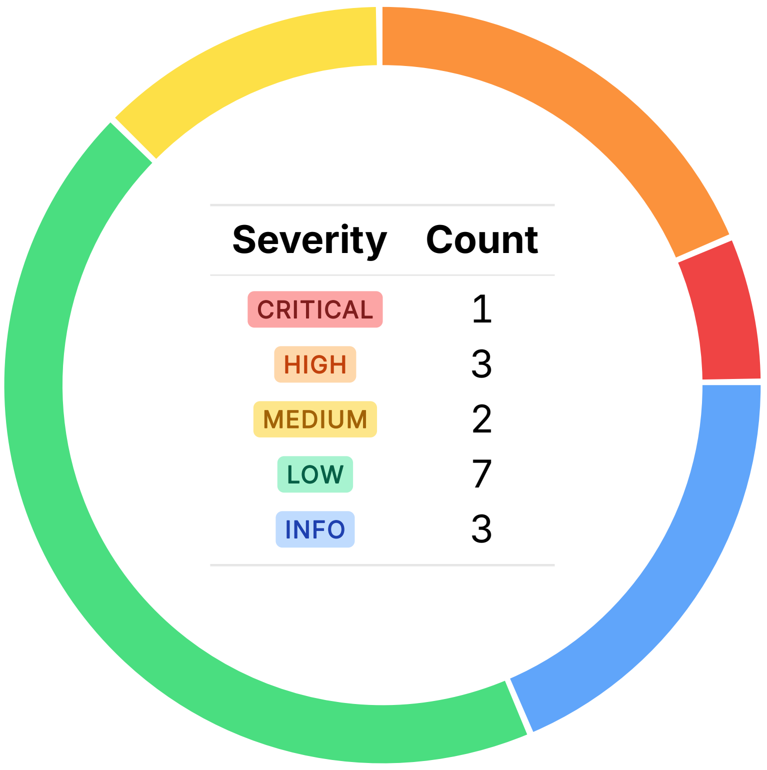
A brief description of the programs is as follows:

Name	Description
factory	It allows for multiple vaults to be deployed dynamically.
vault	It enables users to deposit assets and passively invest them across multiple decentralized finance (DeFi) strategies, such as yield farming, staking, and lending. It automates the management of these assets, offering diversification and exposure to DeFi protocols without the need for active involvement.
strategies	A predefined set of steps for executing investments across one or more protocols, ranging from simple asset holding to complex actions such as yield farming, auto-compounding rewards, or leveraging lending and farming markets.

03 — Findings

Overall, we reported 16 findings.

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



Severity	Count
CRITICAL	1
HIGH	3
MEDIUM	2
LOW	7
INFO	3

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-DIX-ADV-00	CRITICAL	RESOLVED ✓	The vulnerability in the `b_rate` calculation allows attackers to exploit precision loss by manipulating deposits and withdrawals, inflating the `b_rate` and gaining an unfair profit from the vault's share distribution.
OS-DIX-ADV-01	HIGH	RESOLVED ✓	The vault suffers from rounding errors when strategies floor minted shares during deposits and ceil burned shares during withdrawals, resulting in small but consistent losses, which may be exploited to grief or extract value benefiting the depositors.
OS-DIX-ADV-02	HIGH	RESOLVED ✓	`perform_reinvest` sets a minimum output of zero during swaps, allowing attackers to sandwich the transaction and extract value from `BLND` emissions.
OS-DIX-ADV-03	HIGH	RESOLVED ✓	An inflation attack in the `Blend` strategy exploits share price manipulation by the initial depositor, who donates `BLND` tokens to artificially increase share value. As a result, subsequent depositors receive fewer shares due to rounding, leading to a loss of funds.
OS-DIX-ADV-04	MEDIUM	RESOLVED ✓	In the current division logic, while calculating the optimal allocation of amounts for a set of assets, there is a risk of underpayment due to rounding down of the required amount, which may result in users contributing less than expected to the pool.

OS-DIX-ADV-05	MEDIUM	RESOLVED ✓	`fetch_strategy_invested_funds` does not account for the latest balance when locking fees, potentially allowing users to withdraw more tokens than they should by overlooking recent gains and untracked fees.
OS-DIX-ADV-06	LOW	RESOLVED ✓	If `lock_fee` is called when `gains_or_losses` is negative, it effectively reduces the locked fee instead of increasing it.
OS-DIX-ADV-07	LOW	RESOLVED ✓	If a user requests to withdraw more than their available balance, the system silently caps the amount without considering locked fees.
OS-DIX-ADV-08	LOW	RESOLVED ✓	`mint_shares` allows minting zero shares to users, which is unfair as it fails to return any ownership to users who deposit funds. This may result in users effectively losing their deposits.
OS-DIX-ADV-09	LOW	RESOLVED ✓	If the first depositor sets a reserve to zero, subsequent deposits will fail due to the `InsufficientManagedFunds` error, creating a denial-of-service scenario in the vault.
OS-DIX-ADV-10	LOW	RESOLVED ✓	`release_fees` lacks a check for negative amounts, allowing the admin to lock arbitrary fees instead of releasing them.
OS-DIX-ADV-11	LOW	RESOLVED ✓	`deposit` ignores errors from `update_rate` by not propagating them. This implies that an arithmetic error will not abort execution, and the outdated `b_rate` will result in incorrect share calculations.
OS-DIX-ADV-12	LOW	RESOLVED ✓	In `rescue`, the strategy fees are distributed without locking them first, potentially resulting in fee receivers to lose earnings corresponding to recent gains not yet locked for distribution.

bRate Precision Manipulation to Gain Unfair Profit CRITICAL OS-DIX-ADV-00

Description

The vulnerability involves a flaw in the internal `b_rate` calculation in `reserves::update_rate`. In this specific implementation, the `b_rate` is calculated with precision loss ($\frac{underlying\_amount}{b\_tokens}$), implying that it is rounded to a certain number of decimal places (in this case, 9 decimal places). This results in a discrepancy where the internal `b_rate` is always equal to or slightly greater than the actual pool's `b_rate` due to precision loss. This presents an opportunity for attackers to manipulate the rate by depositing and withdrawing specific amounts of tokens to exploit the discrepancy and gain an advantage.

>_ *contracts/strategies/blend/src/reserves.rs* RUST

```rust
pub fn update_rate(
    &mut self,
    underlying_amount: i128,
    b_tokens_amount: i128,
) -> Result<(), StrategyError> {
    // Calculate the new bRate - 9 decimal places of precision
    // Update the reserve's bRate
    let new_rate = underlying_amount
        .fixed_div_floor(b_tokens_amount, SCALAR_9)
        .ok_or_else(|| StrategyError::ArithmeticError)?;
    self.b_rate = new_rate;
    Ok(())
}
```

Proof of Concept

1. The attacker deposits a specially crafted amount of tokens into the strategy, designed to minimize the internal `b_rate` to its lowest possible value to match the blend's `b_rate`.
2. The attacker deposits a large amount into the vault while the `b_rate` is low. With the `b_rate` minimized, the amount of underlying asset returned per share is also low, making the price of each vault share very cheap. This allows the attacker to receive a large number of shares for their deposit.
3. Once the attacker has received a significant number of shares, they reverse their action by making another deposit to push the `b_rate` to its highest possible value.
4. Finally, the attacker withdraws all their shares from the vault. Since the internal `b_rate` is inflated, the underlying asset returned per share is now higher. Thus, the attacker receives more underlying tokens than they originally deposited, resulting in an instant profit.

Remediation

Retrieve the `b_rate` directly from Blend v2 pools instead of computing it internally.

Patch

Fixed in e69f390.

Rounding Discrepancies in Strategy Operations HIGH OS-DIX-ADV-01

Description

The vulnerability arises from rounding inconsistencies in how strategy shares are minted (floored) and burned (ceiled) during deposits and withdrawals. These discrepancies may result in small but consistent value leaks that disproportionately affect remaining vault depositors and may be exploited by the depositor. Thus, when tokens are invested into a strategy, the reported balance increase is typically slightly less than the deposited amount due to rounding errors within the strategy or remote pool interactions, resulting in a minor reported loss, and since investing occurs during deposits and rebalances, the primary issue concerns deposits.

When a user deposits `x` tokens, the vault mints strategy shares based on the full amount that is deposited (`x`). However, due to flooring, the actual amount that gets invested into the strategy is slightly less than `x`. This creates an immediate discrepancy where the vault overestimates its holdings, enabling an attacker to exploit this by repeatedly depositing carefully chosen amounts that maximize this loss, utilizing a looped strategy to grief the vault. Even though, each attack iteration costs the attacker a small amount, the compounded effect erodes vault value.

Similarly, in the case of withdrawals, when `strategy_amount_to_unwind` is unwound from the strategy, the actual decrease in the strategy's balance may exceed the intended amount due to ceiling behavior. However, `cumulative_amount_for_asset` is only increased by the intended `strategy_amount_to_unwind`, not the actual decrease.

Therefore, if `x` tokens are unwound, the strategy may return slightly more than `x` (`y`) due to rounding up, results in a loss of `y - x` within the strategy. The losses will be absorbed by remaining investors (vault shareholders). This case is particularly severe because an attacker may effect substantial vault losses without incurring direct losses themselves. Nevertheless, scaling the exploit will still require multiple transactions and incur additional transaction fees.

Remediation

Refactor the logic to have the strategy always deposit or withdraw optimal amounts. For example, while investing 150 tokens, the strategy may choose to deposit only 100 tokens, such that exactly one share is minted, rendering the remaining 50 tokens idle in the vault until a future investment. Similarly, during the withdrawal of 50 tokens, the strategy may withdraw 100 tokens (burning one full share), sending 50 to the user while retaining the excess in the vault.

Patch

Fixed in 385c939.

Sandwich Attack Due to Lack of Slippage Tolerance HIGH OS-DIX-ADV-02

Description

`blend_pool::perform_reinvest` performs a token swap on Soroswap but sets the minimum amount of received tokens to zero when swapping the emissions for the underlying asset, rendering it vulnerable to a sandwich attack. Thus, before `perform_reinvest` executes, the attacker may frontrun the transaction by buying a large amount of `BLND`, inflating its price.

>_ *contracts/strategies/blend/src/blend_pool.rs* RUST

```rust
pub fn perform_reinvest(e: &Env, config: &Config) -> Result<bool, StrategyError> {
    [...]
    // Swapping BLND tokens to Underlying Asset
    let swapped_amounts = internal_swap_exact_tokens_for_tokens(
        e,
        &blnd_balance,
        &0i128,
        swap_path,
        &e.current_contract_address(),
        &deadline,
        config,
    )?;
    [...]
}
```

Since the minimum received amount is set to zero, the contract is forced to accept a bad rate. Thus, the swap occurs at a manipulated high price for `BLND`, resulting in fewer underlying assets received. Immediately after this swap occurs, the attacker may sell their `BLND`, profiting from the inflated price of `BLND`.

Remediation

Ensure a reasonable slippage tolerance by introducing a valid minimum received amount instead of 0. The expected price may be fetched from an oracle, or a designated manager can be introduced to handle reward harvesting, set the deadline, and specify the minimum acceptable amount.

Patch

Fixed in 645312f.

Share Inflation in Blend Strategy HIGH OS-DIX-ADV-03

Description

Blend strategy is susceptible to an inflation attack which exploits the rounding behavior in share calculations. By artificially inflating the share price, later depositors minted shares will be rounded down, effectively resulting in a loss of their deposit value. The attacker then profits from the lost value. This is possible because the strategy mints shares to represent ownership of underlying assets. The total number of shares and the underlying assets determine the share price.

Proof of Concept

1. Assume a newly deployed `Blend` strategy with no initial deposits, an attacker holding a large amount of `BLND` tokens, and a user planning to deposit assets into the strategy.
2. The attacker deposits underlying assets into the `Blend` strategy and immediately withdraws everything except 1 share. They then donate all `BLND` tokens to the strategy and trigger a harvest. During the harvest, the `BLND` tokens are swapped for underlying assets and reinvested into the `Blend` pool, increasing the total `b_tokens` in the reserve while the total shares remain at 1. This artificially inflates the share price significantly.
3. To maximize the attacker's profit, the strategy must hold a `b_token` balance equal to half the amount the victim is about to mint, plus one. If the victim intends to mint x `b_tokens`, the total `b_tokens` in the strategy should be $x/2+1$.
4. When the user's transaction is processed, they lose approximately 25% of their funds due to rounding down. Before the deposit, the reserves were `total_shares` = 1 and `total_b_tokens` $= x/2+1$, rendering the value of one share $= x/2+1$. After depositing x `b_tokens`, the user's minted shares are calculated as $x/(x/2+1)$, which rounds down to 1, resulting in a loss.
5. At the end of the attack, there are two total shares—one held by the attacker and one by the user—along with $x+x/2+1$ underlying tokens. This results in a new share price of approximately $3x/4$. As a result, the user, who deposited x, has lost 25% of their funds, while the attacker has gained an equivalent amount in profit.

Remediation

Ensure to enforce a minimum share minting requirement.

Patch

Fixed in 5659d64.

Risk of Underpayment via Round Down Division MEDIUM OS-DIX-ADV-04

Description

The issue pertains to the way `calculate_optimal_amounts_and_shares_with_enforced_asset` handles rounding in the division operation while calculating the optimal amounts for assets. The function calculates how much of a particular asset a user should contribute based on their desired `amount` for the enforced asset (`amount_desired_target`) and the reserves of the asset (`reserve`). The calculation utilizes integer division, which by default rounds down. Consequently, the user may end up contributing slightly less than the proportional amount they should contribute. Over time, this will result in significant losses for the pool.

>_ *apps/contracts/vault/src/utils.rs* RUST

```rust
pub fn calculate_optimal_amounts_and_shares_with_enforced_asset([...]) -> (Vec<i128>, i128) {
    [...]
    for (i, asset_allocation) in total_managed_funds.iter().enumerate() {
        [...]
        else {
            // Calculate the proportional allocation for non-enforced assets
            let reserve = asset_allocation.total_amount;
            let amount = reserve
                .checked_mul(amount_desired_target)
                .unwrap_or_else(|| panic_with_error!(e, ContractError::ArithmeticError))
                .checked_div(reserve_target)
                .unwrap_or_else(|| panic_with_error!(e, ContractError::ArithmeticError));
            optimal_amounts.push_back(amount);
        }
    }
    [...]
}
```

Remediation

Ensure to always round up when users are paying into a pool. This guarantees that the pool always receives the correct amount of funds, preventing any shortage or underfunding. Also, round down when calculating the amount to be sent to the user.

Patch

Fixed in d1ec10e.

Inconsistent Fee Locking MEDIUM OS-DIX-ADV-05

Description

In the current implementation, `funds::fetch_strategy_invested_funds` retrieves the strategy's balance and locks the fee only if the `lock_fees` flag is true. When the function is first called, it fetches the current balance of the strategy (`strategy_invested_funds`) and attempts to lock the fees by calling `report.lock_fee(get_vault_fee(e))` . This locks the current vault fee that should be deducted from the balance for reporting purposes.

>_ apps/contracts/vault/src/funds.rs RUST

```rust
pub fn fetch_strategy_invested_funds(e: &Env, strategy_address: &Address, lock_fees: bool) ->
    ↪  Result<i128, ContractError> {
    let strategy_client = get_strategy_client(e, strategy_address.clone());
    let strategy_invested_funds = strategy_client.balance(&e.current_contract_address());

    let mut report = get_report(e, strategy_address);

    if lock_fees {
        report.lock_fee(get_vault_fee(e))?;
        set_report(e, strategy_address, &report);
    }
    Ok(strategy_invested_funds
        .checked_sub(report.locked_fee)
        .unwrap_or(0))
}
```

However, the locked fees may not always be correctly updated in the report, especially if the strategy has experienced gains. `fetch_strategy_invested_funds` does not recalculate the correct fee amount based on the new strategy balance after the gains. Consequently, users may withdraw more claim tokens than they should since the fee for the latest gains is not accounted for.

Remediation

Ensure that the report is updated and fees are recalculated every time the strategy balance is checked.

Patch

Fixed in 9c0d891.

Negative Fees Unlock LOW OS-DIX-ADV-06

Description

`report::lock_fee` does not check whether `gains_or_losses` is positive before calculating and locking fees. If `gains_or_losses` is negative (the strategy experienced a loss instead of a gain), the fee calculation will also yield a negative value. If `gains_or_losses` is negative, the `total_fee` will also be negative. Therfore, `checked_add(total_fee)` reduces the `locked_fee` instead of increasing it, effectively releasing fees instead of locking them.

>_ *apps/contracts/vault/src/report.rs* RUST

```rust
pub fn lock_fee(&mut self, fee_rate: u32) -> Result<(), ContractError> {
    let gains_or_losses = self.gains_or_losses;
    let numerator = gains_or_losses.checked_mul(fee_rate as i128).unwrap();
    let total_fee = numerator.checked_div(MAX_BPS).unwrap();

    self.locked_fee = self.locked_fee.checked_add(total_fee).ok_or(ContractError::Overflow)?;
    self.gains_or_losses = 0;
    Ok(())
}
```

Remediation

Ensure that fees are only locked when `gains_or_losses > 0`.

Patch

Fixed in 5173249.

Withdrawal Limit Violation LOW OS-DIX-ADV-07

Description

The withdrawal process in the Blend Strategy does not enforce strict limits when the requested withdrawal exceeds the user's actual balance. Instead of aborting the transaction, the code simply sets the withdrawal amount to the entire available balance. This behavior is problematic when the withdrawal request is later passed to `unwind_from_strategy` during rebalancing. In `build_actions_from_request`, for a `Withdraw` request, the code checks the number of `bTokens` the user holds.

>_ *blend-contracts/pool/src/pool/actions.rs* RUST

```rust
pub fn build_actions_from_request([...]) -> (Actions, User, bool) {
    [...]
    for request in requests.iter() {
        // verify the request is allowed
        require_nonnegative(e, &request.amount);
        pool.require_action_allowed(e, request.request_type);
        match RequestType::from_u32(e, request.request_type) {
            [...]
            RequestType::WithdrawCollateral => {
                [...]
                if to_burn > cur_b_tokens {
                    to_burn = cur_b_tokens;
                    tokens_out = reserve.to_asset_from_b_token(cur_b_tokens);
                }
               [...]
            }
            [...]
        }
    }[...]
}
```

If the requested amount (converted to `bTokens`) exceeds the user's balance, the function adjusts the amount accordingly. Thus, even if a user requests to withdraw more than their balance, the system silently reduces the withdrawal to the entire available balance.

>_ *apps/contracts/vault/src/strategies.rs* RUST

```rust
pub fn unwind_from_strategy([...]) -> Result<Report, ContractError> {
    let strategy_client = get_strategy_client(e, strategy_address.clone());
    let mut report = get_report(e, strategy_address);
    report.prev_balance =
        ↪ report.prev_balance.checked_sub(*amount).ok_or(ContractError::Underflow)?;
    [...]
}
```

As a result, it is possible to withdraw more than the actual balance when these adjusted amounts are passed to `unwind_from_strategy` during rebalancing in `vault::rebalance`. The function subtracts the requested withdrawal amount from the strategy's `prev_balance`, which will result in it becoming negative.

Remediation

There should be limits on the maximum amount that can be unwound, regardless of the underlying strategy. For example, `locked_fee` should not be unwindable. The maximum amount should be calculated as `report.prev_balance` - `report.locked_fees`.

Patch

Fixed in 341e7d2.

Possibility of Minting Zero Shares to Users LOW OS-DIX-ADV-08

Description

In `deposit::mint_shares` within `vault`, when a user deposits assets into the vault, they should receive vault shares proportional to the value of their deposit. If the number of shares to be minted is zero (`shares_to_mint == 0`), the user will receive no shares despite depositing funds. This creates an unfair situation where the user's assets are locked without any representation of ownership or claim on the vault's assets.

>_ *apps/contracts/vault/src/deposit.rs* RUST

```rust
/// Mint vault shares.
fn mint_shares(
    e: &Env,
    total_supply: &i128,
    shares_to_mint: i128,
    from: Address,
) -> Result<(), ContractError> {
    if *total_supply == 0 {
        if shares_to_mint < MINIMUM_LIQUIDITY {
            panic_with_error!(&e, ContractError::InsufficientAmount);
        }
        internal_mint(e.clone(), e.current_contract_address(), MINIMUM_LIQUIDITY);
        internal_mint(
            e.clone(),
            from.clone(),
            shares_to_mint.checked_sub(MINIMUM_LIQUIDITY).unwrap(),
        );
    } else {
        internal_mint(e.clone(), from, shares_to_mint);
    }
    Ok(())
}
```

Remediation

Check if `shares_to_mint` is zero before any minting takes place.

Patch

Fixed in 96c7da5.

Denial of Service Due to Zero Reserve Amount LOW OS-DIX-ADV-09

Description

In `calculate_optimal_amounts_and_shares_with_enforced_asset`, there is a vulnerability that arises from the condition where the `reserve_target` (the total amount of a specific asset in the pool) is zero. The `reserve_target` check in the function ensures that if the reserve of the enforced asset is zero, the function will panic with an error (`InsufficientManagedFunds`).

>_ *apps/contracts/vault/src/utils.rs* RUST

```rust
pub fn calculate_optimal_amounts_and_shares_with_enforced_asset(
    e: &Env,
    total_managed_funds: &Vec<CurrentAssetInvestmentAllocation>,
    amounts_desired: &Vec<i128>,
    enforced_asset_index: u32,
) -> (Vec<i128>, i128) {
    // Reserve (total managed funds) of the enforced asset
    let reserve_target = total_managed_funds
        .get(enforced_asset_index)
        .unwrap_or_else(|| panic_with_error!(e, ContractError::WrongAmountsLength))
        .total_amount;

    // If reserve target is zero, we cannot calculate the optimal amounts
    if reserve_target == 0 {
        panic_with_error!(e, ContractError::InsufficientManagedFunds);
    }
    [...]
}
```

As a result, in a multi-asset pool, the first depositor may set a zero amount for the first asset, creating a denial-of-service scenario. When subsequent users try to deposit, the function checks the `reserve_target`, which was set to zero by the first depositor, resulting in an immediate panic with `InsufficientManagedFunds`. This issue is temporary, as anyone may resolve it by donating a small amount (even 1 unit) of the enforced asset, which restores the reserve target and allows the pool to function again.

Remediation

Reject deposits that will leave the reserve at zero. If zero-reserve assets are desired, update `calculate_optimal_amounts_and_shares_with_enforced_asset` to skip them and prevent unexpected aborts.

Patch

Fixed in `1b7855d`.

Arbitrary Locking of Fees LOW OS-DIX-ADV-10

Description

`vault::release_fees` currently lacks a check to ensure that the amount parameter is non-negative. This introduces a vulnerability where a negative value may be utilized to manipulate fee balances in unintended ways. Since `amount` represents the amount of fees to be released (unlocked), passing a negative value essentially reverses the operation. Thus, instead of reducing the locked fee balance, it increases it, locking fees rather than releasing them.

>_ *apps/contracts/vault/src/lib.rs* RUST

```rust
fn release_fees(e: Env, strategy: Address, amount: i128) -> Result<Report, ContractError> {
    extend_instance_ttl(&e);
    check_initialized(&e)?;

    let access_control = AccessControl::new(&e);
    access_control.require_role(&RolesDataKey::Manager);

    let mut report = get_report(&e, &strategy);
    report.release_fee(&e, amount)?;
    set_report(&e, &strategy, &report);
    Ok(report)
}
```

Remediation

Assert that `amount` is non-negative before proceeding.

Patch

Fixed in 341e7d2.

Failure to Propagate Error in Rate Update LOW OS-DIX-ADV-11

Description

In the Blend strategy of Defindex, `reserves::deposit` calls `reserves::update_rate` but does not propagate potential errors from it (`?` is missing in the invocation of `update_rate`). `update_rate` performs fixed-point division. If the division fails, the function returns an error (`StrategyError::ArithmeticError`). However, since `deposit` discards the return value after calling `update_rate`, even if it returns an error, the deposit process continues without updating the `b_rate`. As a result, this will affect future share calculations.

>_ *contracts/strategies/blend/src/reserves.rs* RUST

```rust
pub fn update_rate(
    &mut self,
    underlying_amount: i128,
    b_tokens_amount: i128,
) -> Result<(), StrategyError> {
    [...]
    let new_rate = underlying_amount
        .fixed_div_floor(b_tokens_amount, SCALAR_9)
        .ok_or_else(|| StrategyError::ArithmeticError)?;
    [...]
}

pub fn deposit([...]) -> Result<(i128, StrategyReserves), StrategyError> {
    [...]
    let _ = reserves.update_rate(underlying_amount, b_tokens_amount);
    [...]
}
```

Remediation

Utilize `?` in the call to `update_rate` to ensure that if `update_rate` fails, `deposit` will return an error and abort execution.

Patch

Fixed in 983fa7a.

Absence of Fee Locking Before Withdrawal LOW OS-DIX-ADV-12

Description

`Rescue` is a safeguard mechanism that allows the manager or emergency manager to withdraw all funds from a specific strategy and pause it. However, it currently calls `report::distribute_strategy_fees` after asset withdrawal, which may be problematic if fees are not locked beforehand. This sequence may result in incomplete fee distribution, causing fee receivers to lose a portion of their earnings tied to recent gains that have not yet been locked.

>_ *apps/contracts/strategies/blend/src/lib.rs* RUST

```rust
fn rescue(
    e: Env,
    strategy_address: Address,
    caller: Address,
) -> Result<(), ContractError> {
    [...]
    let distribution_result = report::distribute_strategy_fees(&e, &strategy.address,
        ↪  &access_control)?;
    if distribution_result > 0 {
        let mut distributed_fees: Vec<(Address, i128)> = Vec::new(&e);
        distributed_fees.push_back((asset.address.clone(), distribution_result));
        events::emit_fees_distributed_event(&e, distributed_fees.clone());
    }
    [...]
}
```

Remediation

Modify the function to lock fees before distributing.

Patch

Fixed in 341e7d2.

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-DIX-SUG-00	There are several instances where proper validation is not performed, resulting in potential issues.
OS-DIX-SUG-01	Recommendation for modifying the codebase for improved efficiency, functionality, and robustness.
OS-DIX-SUG-02	There are multiple instances of redundant code that should be removed for better maintainability and clarity.

Missing Validation Logic OS-DIX-SUG-00

Description

1. `harvest` emits a harvest event, which includes the `from` address (the caller's address). However, the contract does not verify that this address corresponds to the actual entity invoking the function, allowing anyone to trigger the reward harvesting and reinvestment process from the Blend pool. Ensure that the address passed as `from` in the function is validated against the actual caller of the transaction.

   >_ *apps/contracts/strategies/blend/src/lib.rs* RUST

   ```rust
   fn harvest(e: Env, from: Address) -> Result<(), StrategyError> {
       extend_instance_ttl(&e);
       let config = storage::get_config(&e)?;
       let harvested_blend = blend_pool::claim(&e, &e.current_contract_address(), &config);
       blend_pool::perform_reinvest(&e, &config)?;
       event::emit_harvest(
           &e,
           String::from_str(&e, STARETEGY_NAME),
           harvested_blend,
           from,
       );
       Ok(())
   }
   ```

2. Modify `put_defindex_fee` to validate that the fee is less than or equal to 9,000 (BPS) before allowing it to be set to prevent vault creation from aborting.
3. In the `__constructor`, the factory address is set directly without validation, allowing anyone to deploy a pool and impersonate the legitimate factory. To prevent this, add an authentication check for the factory argument to ensure that only the legitimate factory contract can initialize and deploy the vault.

Remediation

Add the validations stated above.

Patch

1. Fixed in 983fa7a.
2. Fixed in f1eeeed.
3. Fixed in 11122de.

Code Refactoring OS-DIX-SUG-01

Description

1. Since the factory contract handles the deployment of new pools and the WASM hash is tied to the vault's deployment, storing the WASM hash in instance storage rather than persistent storage is more appropriate.
2. In `router::internal_swap_tokens_for_exact_tokens`, an early error should be returned if `amount_in > amount_in_max`, as the function already calculates `amount_in` before the external call. This enhances efficiency by preventing unnecessary contract invocations and improving error handling.

>_ *apps/contracts/vault/src/router.rs* RUST

```rust
pub fn internal_swap_tokens_for_exact_tokens([...]) -> Result<(), ContractError> {
    [...]
    let (reserve_in, reserve_out) = get_reserves_with_pair(
        e.clone(),
        pair_address.clone(),
        token_in.clone(),
        token_out.clone(),
    )?;
    let amount_in = get_amount_in(amount_out.clone(), reserve_in, reserve_out);
    [...]
    let _result: Vec<i128> = e.invoke_contract(
        &get_soroswap_router(e),
        &Symbol::new(&e, "swap_tokens_for_exact_tokens"),
        swap_args.clone(),
    );
    Ok(())
}
```

3. `vault::get_emergency_manager` should explicitly extend the TTL to keep the contract instance alive for a longer duration when called.

Remediation

Incorporate the above-mentioned refactors.

Patch

1. Fixed in c598529.
2. Fixed in 96081b6.
3. Fixed in 341e7d2.

Code Redundancy OS-DIX-SUG-02

Description

1. To maintain internal consistency, the constructor should enforce checks to prevent duplicate assets, as well as duplicate or empty strategies, as these could disrupt the contract's logic.
2. Remove the `bump_instance` calls from the access control storage-related functions (`has_role`, `get_role`, and `set_role`) because the contract instance's TTL (Time-To-Live) is already extended in every entry point function.

>_ *apps/contracts/vault/src/access.rs* RUST

```rust
impl AccessControlTrait for AccessControl {
    fn has_role(&self, key: &RolesDataKey) -> bool {
        bump_instance(&self.0);
        self.0.storage().instance().has(key)
    }

    fn get_role(&self, key: &RolesDataKey) -> Option<Address> {
        bump_instance(&self.0);
        self.0.storage().instance().get(key)
    }

    fn set_role(&self, key: &RolesDataKey, role: &Address) {
        bump_instance(&self.0);
        self.0.storage().instance().set(key, role);
    }
    [...]
}
```

3. Since the `__constructor` now ensures that the vault contract is initialized, calling `check_initialized(&e)?` in other functions is redundant. Previously, this check was utilized to prevent function execution on an uninitialized contract. However, since initialization is now guaranteed at deployment, these checks may be safely removed.

Remediation

Remove the redundant code instances.

Patch

1. Fixed in 341e7d2.
2. Fixed in 5632d2f.
3. Fixed in f1eeeed.

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.