SHERLOCK

Security Review For Burve

Public Best Efforts Audit Contest Prepared For:	**Burve**
Lead Security Expert:	**mstpr-brainbot**
Date Audited:	**April 19 - May 7, 2025**
Final Commit:	**e622ac5**

Introduction

Burve is a 16-token multi-swap for pegged assets launching on Berachain with rehypothecation yields, moving peg handling, an analytic stableswap solution, depeg-protection, and subset-LPing so users can limit themselves to tokens they feel safest in.

Scope

Repository: itos-finance/Burve

Audited Commit: 945f30bfae8afc41af21305ff8c2271ca0ffe6c3

Final Commit: e622ac5666b1b19c5b103df4981b084e8fd43f14

Files:

- src/FullMath.sol
- src/Timed.sol
- src/TransferHelper.sol
- src/integrations/BGTExchange/BGTExchanger.sol
- src/integrations/BGTExchange/IBGTExchanger.sol
- src/integrations/adjustor/DecimalAdjustor.sol
- src/integrations/adjustor/E4626ViewAdjustor.sol
- src/integrations/adjustor/FixedAdjustor.sol
- src/integrations/adjustor/IAdjustor.sol
- src/integrations/adjustor/MixedAdjustor.sol
- src/integrations/adjustor/NullAdjustor.sol
- src/integrations/pseudo4626/noopVault.sol
- src/multi/Adjustor.sol
- src/multi/Asset.sol
- src/multi/Constants.sol
- src/multi/Diamond.sol
- src/multi/Simplex.sol
- src/multi/Store.sol
- src/multi/Token.sol
- src/multi/Value.sol
- src/multi/closure/Closure.sol

- src/multi/closure/Id.sol
- src/multi/facets/LockFacet.sol
- src/multi/facets/SimplexFacet.sol
- src/multi/facets/SwapFacet.sol
- src/multi/facets/ValueFacet.sol
- src/multi/facets/ValueTokenFacet.sol
- src/multi/facets/VaultFacet.sol
- src/multi/vertex/E4626.sol
- src/multi/vertex/Id.sol
- src/multi/vertex/Reserve.sol
- src/multi/vertex/VaultPointer.sol
- src/multi/vertex/VaultProxy.sol
- src/multi/vertex/Vertex.sol
- src/single/Burve.sol
- src/single/Fees.sol
- src/single/IStationProxy.sol
- src/single/Info.sol
- src/single/TickRange.sol

Final Commit Hash

e622ac5666b1b19c5b103df4981b084e8fd43f14

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
9	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0L5S6W
0x23r0
0xNov1ce
0xpinkman
0xpranav
Aamirusmani1552
AshishLac
CasinoCompiler
CodexBugmeNot
Cybrid
Drynooo
Egbe
JeRRy0422
KungFuPanda
RektOracle
Rhaydden
SecSnat
Sparrow_Jac
TessKimy
TheCarrot
Tigerfrake
VinciGearHead
Ziusz
aman
anirruth_
baz1ka
benjamin_0923
bladeee
bretzel
cccz
curly11
elolpuer
future
gh0xt
h2134
heeze
holydevoti0n
imageAfrika
kazan
klaus
kom
m3dython
mstpr-brainbot
newspacexyz
nganhg
ni8mare
peppef
prosper
rsam_eth
shushu
smartkelvin
stonejiajia
teoslaf1
v1c7
vangrim
x0lohaclohell
zark

Issue H-1: Incorrect handling of ERC4626 vaults with fees

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/70

Found by

0L5S6W, 0xNov1ce, CasinoCompiler, Drynooo, aman, bretzel, future, h2134, mstpr-brainbot, newspacexyz, shushu

Summary

When users add liquidity to multi-token pool, the tokens users transfer are deposited to the underlying ERC4626 vaults. These ERC4626 vaults might have deposit and/or withdrawal fees charged.

However, handling of these fees is incorrect in the current implementation and users can avoid these fees. Let's take a look into `addValue` for example:

```
function addValue(
    address recipient,
    uint16 _closureId,
    uint128 value,
    uint128 bgtValue
)
    external
    nonReentrant
    returns (uint256[MAX_TOKENS] memory requiredBalances)
{
    if (value == 0) revert DeMinimisDeposit();
    require(bgtValue <= value, InsufficientValueForBgt(value, bgtValue));
    ClosureId cid = ClosureId.wrap(_closureId);
    Closure storage c = Store.closure(cid);
    uint256[MAX_TOKENS] memory requiredNominal = c.addValue(
        value,
        bgtValue
    );
    // Fetch balances
    TokenRegistry storage tokenReg = Store.tokenRegistry();
    for (uint8 i = 0; i < MAX_TOKENS; ++i) {
        if (!cid.contains(i)) continue; // Irrelevant token.
        address token = tokenReg.tokens[i];
        uint256 realNeeded = AdjustorLib.toReal(
            token,
            requiredNominal[i],
            true
        );
```

```
            requiredBalances[i] = realNeeded;
            TransferHelper.safeTransferFrom(
                token,
                msg.sender,
                address(this),
                realNeeded
            );
            Store.vertex(VertexLib.newId(i)).deposit(cid, realNeeded);
        }
        Store.assets().add(recipient, cid, value, bgtValue);
    }
```

Here, the `realNeeded` amount is the token amount that represents the given `value`, but when these tokens are deposited through `Store.vertex(VertexLib.newId(i)).deposit(cid, realNeeded);`, the fees will get charged that results in less shares being minted. But these fees are not accounted and users' values are added to the pool without these fees.

Root Cause

The root cause of the issue is because the protocol does not transfer enough tokens from users to cover the fees. The affected logic is in ValueFacet contract where users add value to the pool by depositing tokens. Also, it is affected in SwapFacet contract where users transfer tokens to swap into other tokens.

Internal Pre-conditions

An ERC4626 vault with deposit and/or withdrawal fees is used as the underlying vault for the multi-token pool.

External Pre-conditions

None

Attack Path

- Assume there is 1000 assets that represents 1000 value in the multi-token pool.
- Alice deposits 100 value worth of tokens into the pool where she deposits 100 assets.
- As the ERC4626 vault has a 1% deposit fee, 100 assets will be deposited but 99 assets worth of shares will be minted.
- Alice withdraws 100 value worth of tokens from the pool, which returns 100 assets to her.
- The 1 asset is not accounted and is considered as a loss.

Impact

- Users can avoid the fees charged by the ERC4626 vaults, which also can be considered as stealing of the tokens.
- As actual assets is less than the accounted assets, last users will suffer when they withdraw their assets.

PoC

Put the PoC in the `test/facets/Audit.t.sol` file.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;

import {MultiSetupTest} from "./MultiSetup.u.sol";
import {console2 as console} from "forge-std/console2.sol";
import {ValueFacet} from "../../src/multi/facets/ValueFacet.sol";
import {ValueLib} from "../../src/multi/Value.sol";
import {AssetBookImpl} from "../../src/multi/Asset.sol";
import {MAX_TOKENS} from "../../src/multi/Token.sol";
import {MockERC20} from "../mocks/MockERC20.sol";
import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol";

contract MultiTokenAuditTest is MultiSetupTest {

    function setUp() public {
        vm.startPrank(owner);

        _newDiamond();
        _newTokens(3);
        _fundAccount(alice);
        _fundAccount(bob);

        vm.startPrank(owner);
        _initializeClosure(0x03, 1e6 * 1e18); // 0, 1
        _initializeClosure(0x05, 1e6 * 1e18); // 0, 2
        simplexFacet.setClosureFees(0x03, 3 << 123, 0); // 0.3% fee
        simplexFacet.setClosureFees(0x05, 3 << 123, 0); // 0.3% fee

        vm.stopPrank();
    }

    function testAuditERC4626Fee() public {
        uint256 token0BalanceBefore = token0.balanceOf(alice);

        vm.startPrank(alice);
        valueFacet.addValue(alice, 0x03, 100e18, 0);
        valueFacet.removeValue(alice, 0x03, 100e18, 0);
```

```
        vm.stopPrank();

        uint256 token0BalanceAfter = token0.balanceOf(alice);

        assertApproxEqAbs(token0BalanceBefore, token0BalanceAfter, 1e3);
    }
}
```

Update `MockERC4626` to charge fees on deposit, as follows:

```
contract MockERC4626 is ERC4626 {
    constructor(
        ERC20 asset,
        string memory name,
        string memory symbol
    ) ERC20(name, symbol) ERC4626(asset) {}

    function previewDeposit(uint256 assets) public override view returns (uint256) {
        uint256 fee = assets * 1 / 100;
        assets -= fee;
        return _convertToShares(assets, Math.Rounding.Floor);
    }

    function deposit(uint256 assets, address receiver) public override returns
↪   (uint256) {
        uint256 maxAssets = maxDeposit(receiver);
        if (assets > maxAssets) {
            revert ERC4626ExceededMaxDeposit(receiver, assets, maxAssets);
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);

        uint256 fee = assets * 1 / 100;
        ERC20(asset()).transfer(address(0xdead), fee);

        return shares;
    }
}
```

Mitigation

When the ERC4626 vault has deposit and/or withdrawal fees, the protocol should transfer more tokens from users to cover the fees.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/71

Issue H-2: Incorrect implementation of ERC4626View Adjustor

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/113

Found by

Ziusz, bretzel, h2134, newspacexyz

Summary

Adjustor contracts have 2 main functions which are `toNominal` and `toReal`, which convert between real and nominal values of a token.

`ERC4626ViewAdjustor` is an implementation for LSTs such as `stETH`. However, the implementation of `toNominal` and `toReal` is incorrect within this contract. Here's code snippets:

```
function toNominal(
    address token,
    uint256 real,
    bool
) external view returns (uint256 nominal) {
    IERC4626 vault = getVault(token);
    return vault.convertToShares(real);
}

function toReal(
    address token,
    uint256 nominal,
    bool
) external view returns (uint256 real) {
    IERC4626 vault = getVault(token);
    return vault.convertToAssets(nominal);
}
```

The goal of `toNominal` is to convert a real value to a nominal value, such as converting `stETH` balance in `ETH`. In vice versa, `toReal` is to convert a nominal value to a real value, such as converting `ETH` to `stETH`.

But as shown in the code snippets above, the implementation of `toNominal` and `toReal` is reversed, as `toNominal` is using `convertToShares` and `toReal` is using `convertToAssets`.

Root Cause

The root cause of the issue is in ERC4626ViewAdjustor contract where the implementation of `toNominal` and `toReal` is reversed.

Internal Pre-conditions

One of LSTs is used in multi-token pool with `ERC4626ViewAdjustor` as the adjustor.

External Pre-conditions

None

Attack Path

- Assume, `stETH` and `WETH` are used in the pool, and `1 stETH = 1.1 WETH`.
- Alice adds 100 value worth of tokens to the pool, which eventually will require 100 `WETH` and 100 `WETH` worth of `stETH`.
- However, as `toReal` function which is used to calculate the amount of `stETH` uses `convertToAssets`, the amount of `stETH` to deposit becomes `110 stETH` instead of `100/1.1 stETH`.

Impact

Loss of funds for users as they deposit more than they should.

PoC

No response

Mitigation

The fix is to reverse the implementation of `toNominal` and `toReal` in `ERC4626ViewAdjustor` contract.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/itos-finance/Burve/commit/c14995d806cdabc20bfedb094d585ed7b8668c8c

Issue H-3: Incorrect Netting Logic Leads to Excessive Withdrawal Amounts

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/174

Found by

0L5S6W, 0x23r0, 0xpinkman, 0xpranav, KungFuPanda, RektOracle, Rhaydden, Sparrow_Jac, TheCarrot, Tigerfrake, aman, anirruth_, curly11, future, imageAfrika, kazan, m3dython, ni8mare, peppef, prosper, shushu, stonejiajia

Summary

A logical error in the netting calculation will cause incorrect withdrawal amounts as the protocol fails to subtract deposit amounts when netting pending transactions properly.

Root Cause

During the trimBalance, the protocol gets the real balance of a vertex:

```
uint256 realBalance = AdjustorLib.toReal(idx, self.balances[idx], true);
```

which is the `targetReal` and also within the vertex, the protocol gets the actual balance of the closure based on its share balance in the vault:

```
uint256 realBalance = vProxy.balance(cid, false);
```

When the real balance is > targetReal, then we have some residual that needs to be withdrawn from the vault, and the bgtResidual:

```
bgtResidual = FullMath.mulDiv(residualReal, bgtValue, value);
```

becomes the amount to deposit for reserve, thereafter, a commit is made:

```
function trimBalance(
        Vertex storage self,
        ClosureId cid,
        uint256 targetReal,
        uint256 value,
        uint256 bgtValue
    ) internal returns (uint256 reserveSharesEarned, uint256 bgtResidual) {
        VaultProxy memory vProxy = VaultLib.getProxy(self.vid);
        uint256 realBalance = vProxy.balance(cid, false);
        // We don't error and instead emit in this scenario because clearly the
↪  vault is not working properly but if
```

```
        // we error users can't withdraw funds. Instead the right response is to
↪  lock and move vaults immediately.
        if (targetReal > realBalance) {
            emit InsufficientBalance(self.vid, cid, targetReal, realBalance);
            return (0, 0);
        }
        uint256 residualReal = realBalance - targetReal;
        vProxy.withdraw(cid, residualReal);
        bgtResidual = FullMath.mulDiv(residualReal, bgtValue, value);
        reserveSharesEarned = ReserveLib.deposit(
            vProxy,
            self.vid,
            residualReal - bgtResidual
        );
        vProxy.commit();
    }
```

The issue arises in the commit function, at this point both deposit and withdrawal > 0

```
function commit(VaultE4626 storage self, VaultTemp memory temp) internal {
        uint256 assetsToDeposit = temp.vars[1];
        uint256 assetsToWithdraw = temp.vars[2];

        if (assetsToDeposit > 0 && assetsToWithdraw > 0) {
            // We can net out and save ourselves some fees.
            if (assetsToDeposit > assetsToWithdraw) {
                assetsToDeposit -= assetsToWithdraw;
                assetsToWithdraw = 0;
            } else if (assetsToWithdraw > assetsToDeposit) {
                assetsToDeposit = 0;  // @audit
                assetsToWithdraw -= assetsToDeposit;
            } else {
                // Perfect net!
                return;
            }
        }

        if (assetsToDeposit > 0) {
            // Temporary approve the deposit.
            SafeERC20.forceApprove(
                self.token,
                address(self.vault),
                assetsToDeposit
            );
            self.totalVaultShares += self.vault.deposit(
                assetsToDeposit,
                address(this)
            );
            SafeERC20.forceApprove(self.token, address(self.vault), 0);
```

```
        } else if (assetsToWithdraw > 0) {
            // We don't need to hyper-optimize the receiver.
            self.totalVaultShares -= self.vault.withdraw(
                assetsToWithdraw,
                address(this),
                address(this)
            );
        }
    }
```

The code sets assetsToDeposit to 0 and then attempts to subtract this value (now 0) from assetsToWithdraw, resulting in no change to the withdrawal amount.

Internal Pre-conditions

- Both pending deposits (`temp.vars[1] > 0`) and pending withdrawals (`temp.vars[2] > 0`) at the time of commit
- The pending withdrawal amount must be greater than the pending deposit amount (`assetsToWithdraw > assetsToDeposit`)

External Pre-conditions

- There is profit on the vault so preview redeem has increased

Attack Path

- None

Impact

The netting optimization exists to make the protocol more efficient, but this bug prevents it from working properly.

For vaults that charge withdrawal fees (like some yield aggregators), the vault will pay fees on the full withdrawal amount rather than just the net amount.

PoC

No response

Mitigation

```
if (assetsToDeposit > 0 && assetsToWithdraw > 0) {
            if (assetsToDeposit > assetsToWithdraw) {
                assetsToDeposit -= assetsToWithdraw;
                assetsToWithdraw = 0;
            } else if (assetsToWithdraw > assetsToDeposit) {
                assetsToWithdraw -= assetsToDeposit; // audit fix
                assetsToDeposit = 0;
            } else {
                // Perfect net!
                return;
            }
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/itos-finance/Burve/commit/c14995d806cdabc20bfedb094d585ed7b8668c8c

Issue H-4: Incorrect tax distribution when adding value single-sided

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/196

Found by

0xNov1ce, Drynooo, bladeee, future, mstpr-brainbot

Summary

When value is added single-sided to a closure, the LP is required to pay a fee (tax) to the existing LPs. However, the amount is not distributed correctly because valueStaked is updated before the tax is distributed. This causes the new LP to dilute the share of earnings intended for prior participants, leading to unfair reward allocation.

Root Cause

Let's assume a single-sided value is added to a closure. In this case, the `Closure::addValueSingle()` function will be called. This function calculates the `tax` to be paid by the LP and also updates `self.valueStaked`:
https://github.com/sherlock-audit/2025-04-burve/blob/44cba36e2a0c3cd7b6999459bf7746db92f8cc0a/Burve/src/multi/closure/Closure.sol#L208-L209

Afterward, `Closure::addEarnings()` is invoked to distribute the tax earnings to LPs:
https://github.com/sherlock-audit/2025-04-burve/blob/44cba36e2a0c3cd7b6999459bf7746db92f8cc0a/Burve/src/multi/facets/ValueFacet.sol#L132

However, as shown below, the `bgtPerBgtValueX128` and `earningsPerValueX128` values are computed *after* `self.valueStaked` and `self.bgtValueStaked` have been updated by `addValueSingle()`. This means that the new staker is already included in the staked value when earnings are calculated.

As a result, the earnings meant to be distributed to *existing* LPs are significantly diluted, since the new staker (who just paid the tax) also inflates the denominator used in the reward distribution.

```
self.earningsPerValueX128[idx] +=
    (reserveShares << 128) /
    (self.valueStaked - self.bgtValueStaked);
```

This means the protocol is effectively distributing rewards using a denominator that includes the new LP, which reduces the share of existing LPs and undermines fair tax distribution.

Full function here:

https://github.com/sherlock-audit/2025-04-burve/blob/44cba36e2a0c3cd7b6999459bf7746db92f8cc0a/Burve/src/multi/closure/Closure.sol#L662-L703

Internal Pre-conditions

None needed

External Pre-conditions

None needed

Attack Path

Happens naturally when users add liquidity single sided to any closure

Impact

Uneven distribution of fees. Strong disincentive for LP'ers to stake their value. Also, the excess fee is stuck in the contract. Hence, high.

PoC

Textual PoC:

Assume the pool has some initial value (mandatory in deployment) and Alice as the only value provider (LP). Bob comes, and deposits single sided liquidity which will incur fees. This fee should be ideally distributed to the LP'ers which in our case its only Alice. However, the fee is not distributed to Alice. Instead, Alice will only get a small portion of the fees due to wrong calculations.

Coded PoC: In order for this PoC to win add this event to `addValueSingle` in Closure.sol and emit the tax so that we can see what the tax is.

```
event Tapir(uint256 tax);
function addValueSingle(
        address recipient,
        uint16 _closureId,
        uint128 value,
        uint128 bgtValue,
        address token,
        uint128 maxRequired
    ) external nonReentrant returns (uint256 requiredBalance) {
        //..
        emit Tapir(realTax);
    }
```

```
// forge test --match-contract ValueFacetTest --match-test test_tapir -vv
    function test_tapir() public {

        (, , , uint256 valueStaked, ) = simplexFacet.getClosureValue(0x9);
        console.log("valueStaked", valueStaked); // initial valueStaked
        uint256 oneX128 = 1 << 128;
        vm.prank(owner);
        simplexFacet.setClosureFees(0x9, uint128(oneX128 / 100), 0); // One basis
↪  point. Realistic.

        uint128 value = 1e20;
        valueFacet.addValue(alice, 0x9, value, 0);
        uint256[MAX_TOKENS] memory earnings;
        (, , earnings, ) = valueFacet.queryValue(alice, 0x9);
        console.log("earnings[0]", earnings[0]); // expect to be 0
        assertEq(earnings[0], 0);

        // there will be fees which should be distributed to the other LP's.
        address tapir = address(69);
        vm.recordLogs();

        // single value add, fee should be distributed to Alice and the initial
↪  value staked.
        valueFacet.addValueSingle(
            tapir,
            0x9,
            value * 5,
            0,
            tokens[0],
            0
        );

        Vm.Log[] memory entries = vm.getRecordedLogs();
        bytes32 tapirEventSig = keccak256("Tapir(uint256)");

        uint256 tax = 0;
        for (uint256 i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] == tapirEventSig) {
                tax = abi.decode(entries[i].data, (uint256));
                console.log("Tax emitted:", tax);
                break;
            }
        }

        (, , earnings, ) = valueFacet.queryValue(alice, 0x9);
        console.log("earnings[0] after", earnings[0]);  // should be almost the
↪  entire tax above (minus the share of first dead shares)
        (, , earnings, ) = valueFacet.queryValue(tapir, 0x9);
        console.log("earnings[0]", earnings[0]); // should be 0
```

```
        uint256[MAX_TOKENS] memory collectedBalances;
        uint256 collectedBgt;
        vm.prank(Alice);
        (collectedBalances, collectedBgt) = valueFacet.collectEarnings(alice, 0x9);
        console.log("collectedBalances[0]", collectedBalances[0]);
    }
```

Mitigation

add the tax according to previous `valueStaked`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/83

Issue H-5: Attacker captures unclaimed fees by timing deposit with range re-entry and price manipulation

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/288

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

Aamirusmani1552, bladeee, cccz, future, mstpr-brainbot

Summary

Whenever users deposit into or withdraw from Burve, the associated fees are collected and redistributed back into the ranges according to their weights. However, if the current Burve ranges are *out of bounds* relative to the underlying Uniswap V3 pool–and the collected amounts don't meet the minimum liquidity criteria for out-of-range deposits–then no liquidity is compounded. This state persists until the market moves back within the Burve-defined ranges.

An attacker can exploit this by depositing into Burve just before the market re-enters the Burve ranges, then withdrawing shortly after. By sandwiching this moment, the attacker effectively collects a disproportionate share of the accumulated fees.

Root Cause

Assume Burve has a single range that is currently out of range in the underlying Uniswap V3 pool. In this case, only one token–either token0 or token1, depending on the direction the range is out of–will be needed to add liquidity.

In the function collectAndCalcCompound(), if only one token is collected, or if the collected token amount is small enough that dividing it by `amountInUnitLiqX64` results in `0` or a very low value, then the nominal liquidity to be compounded will also be very small–since it's based on the lesser of the two token-derived values.

Once the price in the underlying pool moves back into the Burve-defined range, the other token becomes usable for liquidity provision. An attacker can front-run this moment by depositing into Burve shortly before the range comes back into play–potentially even forcing this through swaps in the underlying pool.

As a result, when `collectAndCalcCompound()` is triggered again(by the withdraw function) and both token amounts are usable, the compounded liquidity will be much higher. The attacker can then immediately withdraw, capturing a disproportionate share of the previously accumulated fees.

Internal Pre-conditions

None needed

External Pre-conditions

1. Underlying pool tick is out of range for all Burve ranges and there are accumulated fees on the other token (the token that is not wanted to add liquidity at current out of range position)

Attack Path

Assume the current Uniswap V3 pool tick is at 100, while Burve has a single active range set between ticks 50–90. Since the range is out of bounds, the position is not earning any liquidity fees. As a result, all of the position's value is held in a single token. In this scenario, only **token0** is required to add liquidity.

Now, assume the following state in Burve's vault in fees:

- `token0` collected: **0**
- `token1` collected: **1000**

Because the Burve range is currently out of range, any deposit or withdrawal will not trigger fee compounding. This happens for reasons explained earlier: the `collectAndCalcCompound()` function cannot add liquidity when it only has the wrong-side token (here, `token1`) available.

This creates a situation where 1000 `token1` is effectively "stuck" in the contract–it won't be converted into liquidity until the underlying pool's price moves back into Burve's specified range (50–90).

An attacker can exploit this by:

1. **Depositing liquidity** into Burve while the range is still out of bounds.
2. **Pushing the Uniswap V3 price down** into Burve's tick range (e.g., from tick 100 to 85) by initiating a large swap.
3. Once the pool price re-enters the Burve range, the next `collectAndCalcCompound()` call will succeed. This time, the previously idle 1000 `token1` will be used to add real liquidity.
4. The attacker then **withdraws** their shares from Burve. Because they entered just before compounding occurred, and now more liquidity was added, their share of the vault is inflated–they effectively captured the full benefit of the previously unclaimed 1000 `token1`.

This results in the attacker siphoning off previously accumulated fees that should have been distributed across all participants proportionally. It's a type of fee-sniping sandwich attack.

Impact

As long as burve ranges are out of range and their accumulated fees on the other token which is always likely, the above attack is possible. Even it's not an attack, users depositing or withdrawing when the range is out of order will not receive fair results.

Additional note, burve ranges can not be changed. Its immutable so it makes the attack more viable and inevitable.

PoC

Described in attack path.

Mitigation

The best mitigation is to make sure the burve ranges are always in range. This can be achieved by adding a small weighted big range.

Issue H-6: Fee Bypass in ValueFacet.removeValueSingle

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/311

Found by

0L5S6W, 0x23r0, AshishLac, CodexBugmeNot, Cybrid, Egbe, JeRRy0422, Rhaydden, SecSnat, Sparrow_Jac, TheCarrot, Tigerfrake, VinciGearHead, aman, anirruth_, baz1ka, benjamin_0923, bladeee, bretzel, curly11, elolpuer, future, gh0xt, heeze, holydevoti0n, imageAfrika, kom, m3dython, prosper, rsam_eth, smartkelvin, v1c7, vangrim

Summary

In the `ValueFacet.removeValueSingle` function, the local variable `removedBalance` is read (as `0`) when computing the real fee (`realTax`), allowing users to withdraw single-token removals without ever paying the protocol fee. This results in a **complete bypass of the intended swap/remove fee**, leading to potential revenue loss.

Root Cause

The function's return variable `removedBalance` is declared but not yet assigned when used to compute `realTax`. The intent was to prorate the nominal fee (`nominalTax`) to "real" token units by computing

```
function removeValueSingle(
    address recipient,
    uint16 _closureId,
    uint128 value,
    uint128 bgtValue,
    address token,
    uint128 minReceive
) external nonReentrant returns (uint256 removedBalance) {
    …
    (uint256 removedNominal, uint256 nominalTax) = c.removeValueSingle(
        value,
        bgtValue,
        vid
    );
    uint256 realRemoved = AdjustorLib.toReal(token, removedNominal, false);
    Store.vertex(vid).withdraw(cid, realRemoved, false);
    // BUG: removedBalance is still zero here
    uint256 realTax = FullMath.mulDiv(
        removedBalance,        // ← should be realRemoved
        nominalTax,
        removedNominal
    );
```

```
    c.addEarnings(vid, realTax);
    removedBalance = realRemoved - realTax;
    require(removedBalance >= minReceive, PastSlippageBounds());
    TransferHelper.safeTransfer(token, recipient, removedBalance);
}
```

https://github.com/sherlock-audit/2025-04-burve/blob/main/Burve/src/multi/facets/ValueFacet.sol#L214-L245

```
realTax = realRemoved * nominalTax  / removedNominal;
```

but instead `removedBalance` (zero) is used as the numerator, making `realTax == 0` always.

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

N/A

Impact

The protocol loses **100% of the intended fees** on every single token removal.

PoC

No response

Mitigation

Replace the incorrect use of `removedBalance` with `realRemoved` when computing `realTax`. Specifically:

```
-    uint256 realTax = FullMath.mulDiv(
-        removedBalance,
-        nominalTax,
-        removedNominal
-    );
+    // Compute the real-world fee based on the actual tokens removed
```

```
+    uint256 realTax = FullMath.mulDiv(
+        realRemoved,
+        nominalTax,
+        removedNominal
+    );
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/itos-finance/Burve/commit/ca58c88be97e8f3f7a4617b171377d77fca52cdc

Issue H-7: An attacker can drain assets from a Closure by exploiting the NoopVault via a donation attack

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/387

Found by

TessKimy, m3dython, x0lohaclohell

Summary

The absence of protective mechanisms in the NoopVault as a vertex vault allows an attacker to exploit the ERC4626 standard's vulnerability to donation attacks. By front-running the first legitimate deposit and donating assets directly(without interacting with the valueFacet) to the vault without receiving shares, the attacker can manipulate the share-to-asset ratio. This manipulation enables the attacker to withdraw a disproportionate amount of assets, effectively draining each vertex in a Closure.

Root Cause

There is no mechanism to prevent donation attacks, such as initializing the vault with a non-zero share supply or implementing virtual assets/shares. This omission allows an attacker to manipulate the vault's share-to-asset ratio by donating assets without receiving corresponding shares.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;
import "openzeppelin-contracts/token/ERC20/extensions/ERC4626.sol";
import "openzeppelin-contracts/token/ERC20/ERC20.sol";

contract NoopVault is ERC4626 {
    constructor(
        ERC20 asset,
        string memory name,
        string memory symbol
    ) ERC20(name, symbol) ERC4626(asset) {}
}
```

Internal Pre-conditions

1. Admin adds a new NoopVault as a vertex in the Burve protocol.
2. The vault has zero total supply and total assets upon initialization.

3. No initial deposit is made to the vault to set a baseline share-to-asset ratio.

External Pre-conditions

1. Berachain, being an L1 blockchain, has a transparent mempool where pending transactions can be observed.
2. An attacker monitors the mempool for transactions adding new vertices (vaults) to the Burve protocol.

Attack Path

1. Attacker observes a transaction in the mempool adding a new Vault as a vertex.
2. Before any legitimate user adds value through the valueFacet, the attacker front-runs by depositing a minimal amount (e.g., 1 wei) directly into the vault, receiving all the initial shares.
3. The attacker then donates a significant amount of assets directly to the vault without minting new shares, inflating the total assets.
4. A legitimate user deposits assets into the vault, expecting to receive shares proportional to their deposit. However, due to the inflated asset pool and the attacker's control of all shares, the user receives fewer shares than expected.
5. The attacker redeems their shares, withdrawing a disproportionate amount of assets from the vault, effectively draining the Closure.

Impact

Users suffer significant losses as their deposits yield fewer shares than expected, leading to a loss in asset value and the attacker gains assets equivalent to the legitimate users deposits without providing corresponding value. This attack can be cordinated to target all vertices in a closure and drain the whole closure.

PoC

No response

Mitigation

Implement Virtual Assets/Shares: Incorporate virtual assets and shares in the vault's accounting to simulate an initial balance, mitigating the impact of donation attacks.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/79

Issue H-8: ValueFacet::removeValueSingle(...) will withdraw less than required from the vertex vault due to unaccounted tax

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/406

Found by

0L5S6W, 0xNov1ce, Cybrid, TessKimy, Tigerfrake, VinciGearHead, bladeee, bretzel, elolpuer, heeze, nganhg, v1c7

Summary

NB: This issue assumes that issue `removeValueSingle(...) will always produce a zero realTax, leading to issues` has been fixed

When users want to remove value from a single vertex in a closure with `removeValueSingle`, the `Burve` protocol charges a tax to prevent users from using add value and remove value as a swapping mechanism. When the protocol determines the required token amount and tax to be removed based on the input value, it then withdraws the set amount from the underlying vertex vault in order to deposit tax earnings and transfer user funds. However, there seems to be a mismatch between the amount being removed and the amount being withdrawn, as the calculated tax is not being withdrawn from the vault.

Root Cause

When the removedAmount is calculated in `Closure::removeValueSingle(...)`, it has the tax deducted from it before being returned. However, when withdrawing from the vault in the facet, this tax is not taken into account, leading to fewer tokens being withdrawn from the vault.

Internal Pre-conditions

1. The user adds value to a closure with a base fee.
2. The user wants to remove value for a single token.

External Pre-conditions

N/A

Attack Path

N/A

Impact

In the current state, this mismatch is compensated, as the user is being double taxed here. However, if the issue `Users will receive fewer tokens when using ValueFacet::removeValueSingle(...) due to double tax` is fixed, then because fewer funds are taken out from the vault, there won't be enough to cover the user transfer, leading to a revert due to insufficient balance and thus to DoS of the `removeValueSingle(...)` function.

PoC

N/A

Mitigation

Calculate the real tax before withdrawing from the vault, and when withdrawing, add the `realTax` to the `realRemoved` being withdrawn.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/itos-finance/Burve/commit/ca58c88be97e8f3f7a4617b171377d77fca52cdc

Issue H-9: Reserve Share Overflows Due to Too Strict Reward Calculation Mechanism

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/452

Found by

TessKimy

Summary

In the system's vertex balancing logic, `trimBalance()` withdraws surplus funds from a vertex's vault and deposits them into the global reserve. The reserve maintains its own share accounting per token, and share minting is based on proportional value derived from vault balances.

Due to **precision-sensitive arithmetic and lack of lower bounds**, the deposit logic is vulnerable to **silent rounding attacks** that cause `reserve.shares[idx]` to inflate gradually or exponentially. Repeated calls to `trimBalance()` (it's not public we will call it with removeValue for trimming) with crafted values can amplify this inflation and potentially lead to **overflow of the share counter**, destabilizing the reserve's accounting and breaking value logic across closures.

Root Cause

In `trimBalance()`:

```
uint256 residualReal = realBalance - targetReal;
...
reserveSharesEarned = ReserveLib.deposit(
    vProxy,
    self.vid,
    residualReal - bgtResidual
);
```

If `bgtResidual - residualReal` is so close to zero, then the deposit call receives a very small nonzero amount–**insufficient to affect the vault's balance** meaningfully but still capable of minting shares due to rounding.

In ReserveLib.deposit():

```
shares = (balance == 0)
    ? amount * SHARE_RESOLUTION
    : (amount * reserve.shares[idx]) / balance;
```

When `amount > 0` but `balance 0`, the share calculation becomes:

$$\text{shares} = \frac{\text{amount} \cdot \text{reserve.shares[idx]}}{\text{balance}} \rightarrow \text{inflates rapidly}$$

This allows the attacker to:

1. Induce tiny residuals repeatedly using trim cycles.
2. Cause share minting to diverge and grow exponentially due to division by near-zero balances.
3. Eventually **overflow the `reserve.shares[idx]` counter**, destabilizing reserve logic.

Internal Pre-conditions

No need

External Pre-conditions

No need

Attack Path

Attack is very easy to understand in PoC

Impact

- **Total share overflow**, leading to irreversible corruption of internal accounting.
- **Silent amplification** through repeated small trims – no reverts occur due to the design relying on best-effort balance corrections.

Because the rounding behavior does not revert, the issue can be triggered with minimal input. It reverts on trim balance calls. Therefore, none of the function can work anymore because it will always fail when we try to deposit that balance to reserve.

PoC

You can test it by following PoC test with full setup For better visibility add console.log to ReserveLib::deposit function

```
shares = (balance == 0)
    ? amount * SHARE_RESOLUTION
    : (amount * reserve.shares[idx]) / balance;
console.log("Reserve shares: ", shares);
```

```
reserve.shares[idx] += shares;
console.log("Reserve shares total: ", reserve.shares[idx])
```

Full PoC Setup

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../src/multi/Diamond.sol";
import "../src/multi/facets/LockFacet.sol";
import "../src/multi/facets/SimplexFacet.sol";
import "../src/multi/facets/SwapFacet.sol";
import "../src/multi/facets/ValueFacet.sol";
import "../src/multi/facets/ValueTokenFacet.sol";
import "../src/multi/facets/VaultFacet.sol";
import "../src/integrations/adjustor/DecimalAdjustor.sol";
import "./MockERC20.sol";
import "../src/integrations/pseudo4626/noopVault.sol";

contract BaseTest is Test {
    SimplexDiamond dia;
    address diaA;
    ERC20 USDC;
    ERC20 DAI;
    uint16 _cid;
    NoopVault usdc_vault;
    NoopVault dai_vault;
    function setUp() public {

        // Deploy facets
        address simplexFacet = address(new SimplexFacet());
        address swapFacet = address(new SwapFacet());
        address valueFacet = address(new ValueFacet());
        address valueTokenFacet = address(new ValueTokenFacet());
        address vaultFacet = address(new VaultFacet());

        USDC = new MockERC20("USDC", "USDC", 1_000_000_000_000e6, 6);
        DAI = new MockERC20("DAI", "DAI", 1_000_000_000_000e18, 18);

        address adjustor = address(new DecimalAdjustor());
        BurveFacets memory facets = BurveFacets({
            valueFacet: valueFacet,
            valueTokenFacet: valueTokenFacet,
            simplexFacet: simplexFacet,
            swapFacet: swapFacet,
            vaultFacet: vaultFacet,
            adjustor: adjustor
```

```
        });

        dia = new SimplexDiamond(facets, "Burve", "BRV");
        diaA = address(dia);

        usdc_vault = new NoopVault(USDC, "USDC Vault", "USDCV");
        dai_vault = new NoopVault(DAI, "DAI Vault", "DAIV");

        VaultType usdc_vault_type = VaultType.E4626;
        VaultType dai_vault_type = VaultType.E4626;

        string memory signature = "addVertex(address,address,uint8)";
        bytes memory argsUSDC = abi.encode(address(USDC), address(usdc_vault),
↪   usdc_vault_type);
        bytes memory argsDAI = abi.encode(address(DAI), address(dai_vault),
↪   dai_vault_type);

        diaA.call(encodeCall(signature, argsUSDC));
        diaA.call(encodeCall(signature, argsDAI));

        deal(address(USDC), address(this), 1_000_000e6);
        deal(address(DAI), address(this), 1_000_000e18);

        USDC.approve(diaA, type(uint256).max);
        DAI.approve(diaA, type(uint256).max);

        _cid = 0x0003;
        uint128 startingTarget = 1e12;
        uint128 baseFeeX128 = 0;
        uint128 protocolTakeX128 = 0;

        signature = "addClosure(uint16,uint128,uint128,uint128)";
        bytes memory args = abi.encode(_cid, startingTarget, baseFeeX128,
↪   protocolTakeX128);
        diaA.call(encodeCall(signature, args));

    }
    function testReserve() public {
        address bob = address(0xB0B);
        address whale = address(0x11);

        deal(address(USDC), bob, 1_000_000e6);
        deal(address(DAI), bob, 1_000_000e18);
        deal(address(USDC), whale, 10_000_000e6);
        deal(address(DAI), whale, 10_000_000e18);
        vm.startPrank(whale);

        USDC.approve(diaA, type(uint256).max);
        DAI.approve(diaA, type(uint256).max);
```

```
        string memory addValueSignature =
↪  "addValue(address,uint16,uint128,uint128)";
        bytes memory addValue1Million = abi.encode(whale, _cid, 2_000_000e18, 0);

        (bool sc5, bytes memory d5) = diaA.call(encodeCall(addValueSignature,
↪  addValue1Million));
        assert(sc5);

        vm.startPrank(bob);

        USDC.approve(diaA, type(uint256).max);
        DAI.approve(diaA, type(uint256).max);

        bytes memory addValue450 = abi.encode(bob, _cid, 450, 0);

        (bool sc, bytes memory d) = diaA.call(encodeCall(addValueSignature,
↪  addValue450));

        assert(sc);
        DAI.transfer(address(dai_vault), 4);

        console.log("\nRemove value part");

        for(uint256 i = 0; i < 450; i++){
            console.log("\n\n");
            string memory removeValueSignature =
↪  "removeValue(address,uint16,uint128,uint128)";
            bytes memory removeValueDust = abi.encode(bob, _cid, 1, 0);

            (bool sc2, bytes memory d2) =
↪  diaA.call(encodeCall(removeValueSignature, removeValueDust));
            // Don't need to check success, most probably it will revert
            // for some of them but it doesn't matter
        }


        vm.stopPrank();

        vm.startPrank(whale);

        string memory removeValueSignature =
↪  "removeValue(address,uint16,uint128,uint128)";
        bytes memory removeValue1million = abi.encode(whale, _cid, 2_000_000e18, 0);

        (bool sc3, bytes memory d3) = diaA.call(encodeCall(removeValueSignature,
↪  removeValue1million));
        assert(sc3);


    }
```

```
}
```

Output

It will fail with following error:

```
            [0] console::log("Reserve shares: ",
↪  459559194249870274726382384477933621776054052562004428510610236767724801592 66
↪  [4.595e76]) [staticcall]
                ← [Stop]
            ← [Revert] panic: arithmetic underflow or overflow (0x11)
        ← [Revert] panic: arithmetic underflow or overflow (0x11)
    ← [Revert] panic: assertion failed (0x01)

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 317.06ms (305.83ms
↪  CPU time)

Ran 1 test suite in 2.19s (317.06ms CPU time): 0 tests passed, 1 failed, 0 skipped
↪  (1 total tests)

Failing tests:
Encountered 1 failing test in test/BaseTest.t.sol:BaseTest
[FAIL: panic: assertion failed (0x01)] testReserve() (gas: 83890019)
```

Mitigation

Apply minimum balance difference between recorded balance and gathered balance before triggering trim balance

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/78

Issue M-1: User can backrun an admin calling setEX128 and steal the difference in tokens

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/120

Found by

0xNov1ce, TessKimy, bladeee

Summary

The formula for calculating the value of a given token is:

`(e+2)t - (e+1)^2 * t^2 / (x + et)`

Where `e` is the efficiency factor. To maintain a balanced pool, `v(tokens)` should equal `target * n`. However, changing `e` for a specific token does **not** trigger a recomputation of `target * n`. This oversight allows users to exploit the system:

- They can **steal** the difference via `removeTokenForValue`, or
- **Add** the difference for free via `addTokenForValue`.

```
//@review - User crafts a specific amount so that `newTargetX128` matches the
↪  current `targetX128`
self.balances[idx] -= taxedRemove;

uint256 newTargetX128;
{
    (uint256[] memory mesX128, uint256[] memory mxs) = ValueLib.stripArrays(self.n,
↪  esX128, self.balances);
    newTargetX128 = ValueLib.t(
        searchParams,
        mesX128,
        mxs,
        self.targetX128
    );
}
//@review - After recalculating `t` with the new `ex` and updated balance,
↪  `newTargetX128` matches `self.targetX128`, enabling the user to spend
↪  negligible amounts (dust) and withdraw excess tokens
uint256 valueX128 = ((self.targetX128 - newTargetX128) * self.n);
value = valueX128 >> 128;
if ((value << 128) > 0) value += 1;
```

Root Cause

The pool owner can modify `e` via the `SimplexFaces::setEX128` function:

```
function setEX128(address token, uint256 eX128) external {
    AdminLib.validateOwner();
    uint8 idx = TokenRegLib.getIdx(token);
    emit EfficiencyFactorChanged(msg.sender, token, SimplexLib.getEX128(idx),
↪   eX128);
    SimplexLib.setEX128(idx, eX128);
}
```

After changing `eX128`, the system **does not recompute** `target * n`. This flaw permits users to:

- [addSingleForValue](https://github.com/sherlock-audit/2025-04-burve/blob/main/Burve/src/multi/facets/ValueFacet.sol#L139-L172)
- [removeSingleForValue](https://github.com/sherlock-audit/2025-04-burve/blob/main/Burve/src/multi/facets/ValueFacet.sol#L248-L277)

thus extracting tokens without providing equivalent value.

Internal Pre-conditions

1. Admin modifies `e` by either increasing or decreasing it – a likely action since `e` determines token concentration in the pool.

External Pre-conditions

None.

Attack Path

1. Admin increases `e` for a token.
2. A user backruns the transaction and calls `removeTokenForValue`, choosing an `amount` that ensures `newTargetX128` equals the original `targetX128`.
3. The user spends minimal value and collects the excess tokens.

Impact

Instead of the excess tokens moving to the reserve as intended, malicious users can seize them freely. This leads to:

- Unauthorized asset extraction

- Potential destabilization of the pool
- Arbitrage opportunities at the expense of the protocol

PoC

None provided.

Mitigation

Upon calling `setEX128`, **immediately adjust balances** to preserve `targetX128`:

- **If `e` is increased**: move the corresponding value to the reserve.
- **If `e` is decreased**: inject the additional value accordingly.

This ensures the pool remains properly balanced and prevents malicious exploitation.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/73

Issue M-2: Simplex ownership cannot be transferred

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/130

Found by

0L5S6W, CasinoCompiler, TessKimy, Tigerfrake, aman, h2134, heeze, imageAfrika, klaus, kom, teoslaf1

Summary

`acceptOwnership()` is missed when adding facets to SimplexDiamond, leading to the ownership cannot be transferred.

Root Cause

When creates `SimplexDiamond`, BaseAdminFacet is added to the diamond.

SimplexDiamond::constructor():

```
        {
            bytes4[] memory adminSelectors = new bytes4[](3);
@>          adminSelectors[0] = BaseAdminFacet.transferOwnership.selector;
@>          adminSelectors[1] = BaseAdminFacet.owner.selector;
@>          adminSelectors[2] = BaseAdminFacet.adminRights.selector;
            cuts[2] = FacetCut({
                facetAddress: address(new BaseAdminFacet()),
                action: FacetCutAction.Add,
                functionSelectors: adminSelectors
            });
        }
```

`transferOwnership()` in `BaseAdminFacet` does not transfer the ownership directly to the new owner, it requires the new pending owner to accept the ownership.

BaseAdminFacet::transferOwnership():

```
function transferOwnership(address _newOwner) external override {
    AdminLib.reassignOwner(_newOwner);
}
```

BaseAdminFacet::acceptOwnership():

```
/// The pending owner can accept their ownership rights.
function acceptOwnership() external {
    emit IERC173.OwnershipTransferred(AdminLib.getOwner(), msg.sender);
```

```
    AdminLib.acceptOwnership();
}
```

The problem is that `acceptOwnership()` is not added to the cut, as a result, the ownership won't be accepted.

Internal Pre-conditions

Simplex owner transfers the ownership.

External Pre-conditions

None

Attack Path

None

Impact

The new pending owner won't be able to accept the ownership.

PoC

Please run `forge test --mt testAudit_SimplexOwnershipCannotBeTransferred`.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;

import "forge-std/Test.sol";
import "../src/multi/Diamond.sol";
import "../src/integrations/adjustor/MixedAdjustor.sol";

contract AuditTest is Test {

    address owner = makeAddr("OWNER");

    SimplexDiamond simplex;

    function setUp() public {

        BurveFacets memory facets;

        {
            address vauleFacet = address(new ValueFacet());
```

```
            address valueTokenFacet = address(new ValueTokenFacet());
            address simplexFacet = address(new SimplexFacet());
            address swapFacet = address(new SwapFacet());
            address vaultFacet = address(new VaultFacet());
            MixedAdjustor mixedAdjustor = new MixedAdjustor();
            address adjustor = address(mixedAdjustor);

            facets = BurveFacets({
                valueFacet: vauleFacet,
                valueTokenFacet: valueTokenFacet,
                simplexFacet: simplexFacet,
                swapFacet: swapFacet,
                vaultFacet: vaultFacet,
                adjustor: adjustor
            });
        }

        vm.prank(owner);
        simplex = new SimplexDiamond(facets, "ValueToken", "BVT");
    }

    function testAudit_SimplexOwnershipCannotBeTransferred() public {
        address newOwner = makeAddr("NEW OWNER");

        bool success;
        bytes memory returnData;

        vm.prank(owner);
        (success, ) = address(simplex).call(
            abi.encodeWithSelector(BaseAdminFacet.transferOwnership.selector,
↪   newOwner)
        );
        require(success, "Transfer Ownership failed");

        vm.prank(newOwner);
        (success, returnData) = address(simplex).call(
            abi.encodeWithSelector(BaseAdminFacet.acceptOwnership.selector)
        );
        // Ownership cannot be accepted
        assertFalse(success);

        (success, returnData) = address(simplex).call(
            abi.encodeWithSelector(BaseAdminFacet.owner.selector)
        );
        require(success, "Get owner failed");

        // The owner does not change
        address currentOwner = abi.decode(returnData, (address));
        assertEq(currentOwner, owner);
    }
```

```
}
```

Mitigation

Add `acceptOwnership()` to admin cut.

```
        {
-           bytes4[] memory adminSelectors = new bytes4[](3);
+           bytes4[] memory adminSelectors = new bytes4[](4);
            adminSelectors[0] = BaseAdminFacet.transferOwnership.selector;
            adminSelectors[1] = BaseAdminFacet.owner.selector;
            adminSelectors[2] = BaseAdminFacet.adminRights.selector;
+           adminSelectors[3] = BaseAdminFacet.acceptOwnership.selector;
            cuts[2] = FacetCut({
                facetAddress: address(new BaseAdminFacet()),
                action: FacetCutAction.Add,
                functionSelectors: adminSelectors
            });
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/77

Issue M-3: Protocol fee resides in the diamond contract can be wrongly sent to users if the underlying vault temporarily disables withdrawal

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/247

Found by

h2134

Summary

Protocol fee resides in diamond contract can be wrongly sent to users if the underlying vault temporarily disables withdrawal.

Root Cause

When user collects earnings, `trimAllBalances()` is called to catch up on rehypothecation gains.

ValueFacet::collectEarnings():

```
// Catch up on rehypothecation gains before we claim fees.
Store.closure(cid).trimAllBalances();
```

The earnings are to be withdrawn from and then deposited back into the vault.

VertexImpl::trimBalance():

```
        uint256 residualReal = realBalance - targetReal;
@>      vProxy.withdraw(cid, residualReal);
        bgtResidual = FullMath.mulDiv(residualReal, bgtValue, value);
@>      reserveSharesEarned = ReserveLib.deposit(
            vProxy,
            self.vid,
            residualReal - bgtResidual
        );
        vProxy.commit();
```

Normally, the withdrawn amount and deposited amount are expected to have a perfect net so there would not be an actual deposit or withdrawal.

VaultE4626Impl::commit():

```
        if (assetsToDeposit > 0 && assetsToWithdraw > 0) {
            // We can net out and save ourselves some fees.
            if (assetsToDeposit > assetsToWithdraw) {
                assetsToDeposit -= assetsToWithdraw;
                assetsToWithdraw = 0;
            } else if (assetsToWithdraw > assetsToDeposit) {
                assetsToDeposit = 0;
                assetsToWithdraw -= assetsToDeposit;
            } else {
                // Perfect net!
@>              return;
            }
        }
```

However, things would go wrong if the underlying vault temporarily disables withdrawal. As can be seen, `VaultProxy` queries the withdrawable amount from the underlying vault, this is done by calling `maxWithdraw()` of the underlying vault.

VaultProxyImpl::withdraw():

```
    function withdraw(
        VaultProxy memory self,
        ClosureId cid,
        uint256 amount
    ) internal {
        // We effectively don't allow withdraws beyond uint128 due to the capping
 ↪  in balance.
        uint128 available = self.active.balance(cid, false);
@>      uint256 maxWithdrawable = self.active.withdrawable();
        if (maxWithdrawable < available) available = uint128(maxWithdrawable);

        if (amount > available) {
            self.active.withdraw(cid, available);
            self.backup.withdraw(cid, amount - available);
        } else {
            self.active.withdraw(cid, amount);
        }
    }
```

VaultE4626Impl::withdrawable():

```
    /// Return the most we can withdraw right now.
    function withdrawable(
        VaultE4626 storage self
    ) internal view returns (uint128) {
@>      return min128(self.vault.maxWithdraw(address(this)));
    }
```

When the underlying vault disables withdrawal, `maxWithdraw()` returns 0. Hence the `VaultProxy` withdrawal would return early without queueing withdrawal and updating `VaultTemp.vars[2]` (assets to withdraw).

```
    /// Queue up a withdrawal for a given cid.
    function withdraw(
        VaultPointer memory self,
        ClosureId cid,
        uint256 amount
    ) internal {
@>      if (isNull(self) || amount == 0) return;

        if (self.vType == VaultType.E4626) {
            getE4626(self).withdraw(self.temp, cid, amount);
        } else {
            revert VaultTypeUnrecognized(self.vType);
        }
    }
```

As a result, when the queued deposit/withdrawal is committed, there is not prefect net between `assetsToDeposit` and `assetsToWithdraw`, **hence the tokens in the diamond contract will be deposited to the underlying vault and the transferred amount is `assetsToDeposit`.**

By then `trimBalance()` is done and the process flow back into `collectEarnings()`, the `withdraw()` in `ReserveLib` is called.

ValueFacet::collectEarnings():

```
collectedBalances[i] = ReserveLib.withdraw(
    vid,
    collectedShares[i]
);
```

ReserveLib::withdraw():

```
    function withdraw(
        VertexId vid,
        uint256 shares
    ) internal returns (uint256 amount) {
        Reserve storage reserve = Store.reserve();
        uint8 idx = vid.idx();
        if (reserve.shares[idx] == 0) return 0;
        VaultProxy memory vProxy = VaultLib.getProxy(vid);
        uint128 balance = vProxy.balance(RESERVEID, true);
        amount = (shares * balance) / reserve.shares[idx];
@>      vProxy.withdraw(RESERVEID, amount);
@>      vProxy.commit();
        reserve.shares[idx] -= shares;
```

```
    }
```

Likewise, `VaultProxy` returns earlier when the underlying vault's `maxWithdraw()` returns 0, and there is no actual withdrawal in `commit()` as `assetsToWithdraw` is 0.

Then back in `collectEarnings()`, **it sends tokens to the user by transferring the tokens in the diamond contract (again)**.

ValueFacet::collectEarnings():

```
TransferHelper.safeTransfer(
    tokenReg.tokens[i],
    recipient,
    collectedBalances[i]
);
```

The tokens reside in diamond contract are the protocol fee, but they are wrongly sent to users.

Internal Pre-conditions

There are protocol fee in the diamond contract.

External Pre-conditions

The underlying vault disable withdrawal temporarily.

Attack Path

None

Impact

The protocol fee is lost.

PoC

Please run `forge test --mt testAudit_CollectingEarningsEatsUpProtocolFee`.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;

import "forge-std/Test.sol";
import "./mocks/MockERC20.sol";
import "openzeppelin-contracts/mocks/docs/ERC4626Fees.sol";
```

```
import {VaultType} from "../src/multi/vertex/VaultPointer.sol";
import "../src/multi/Diamond.sol";
import "../src/integrations/adjustor/MixedAdjustor.sol";
import "../src/integrations/adjustor/DecimalAdjustor.sol";

contract AuditTest is Test {
    uint8 constant MAX_TOKENS = 16;

    address owner = makeAddr("OWNER");
    address diamond;

    MockERC20 tokenA = new MockERC20("Token A", "A", 18);
    MockERC20 tokenB = new MockERC20("Token B", "B", 6);

    MockERC4626WithFees vaultA = new MockERC4626WithFees(tokenA, "Vault A", "VA");
    MockERC4626WithFees vaultB = new MockERC4626WithFees(tokenB, "Vault B", "VB");

    function setUp() public {
        vm.startPrank(owner);

        BurveFacets memory facets;

        {
            address vauleFacet = address(new ValueFacet());
            address valueTokenFacet = address(new ValueTokenFacet());
            address simplexFacet = address(new SimplexFacet());
            address swapFacet = address(new SwapFacet());
            address vaultFacet = address(new VaultFacet());
            MixedAdjustor mixedAdjustor = new MixedAdjustor();
            // Configure adjustor
            {
                DecimalAdjustor decimalAdjustor = new DecimalAdjustor();
                mixedAdjustor.setAdjustor(address(tokenB),
↪   address(decimalAdjustor));
            }
            address adjustor = address(mixedAdjustor);

            facets = BurveFacets({
                valueFacet: vauleFacet,
                valueTokenFacet: valueTokenFacet,
                simplexFacet: simplexFacet,
                swapFacet: swapFacet,
                vaultFacet: vaultFacet,
                adjustor: adjustor
            });
        }

        diamond = address(new SimplexDiamond(facets, "ValueToken", "BVT"));

        vm.label(diamond, "Diamond");
```

```
        vm.label(address(tokenA), "Token A");
        vm.label(address(tokenB), "Token B");
        vm.label(address(vaultA), "Vault A");
        vm.label(address(vaultB), "Vault B");

        // Add Vertex
        {
            // TokenA
            SimplexFacet(diamond).addVertex(address(tokenA), address(vaultA),
↪   VaultType.E4626);

            // TokenB
            SimplexFacet(diamond).addVertex(address(tokenB), address(vaultB),
↪   VaultType.E4626);
        }

        // Add Closure
        {
            bool success;

            uint16 cid = 3; // 0b11
            uint128 startingTarget = 1e12;
            uint128 baseFeeX128 = 0;
            uint128 protocolTakeX128 = 0;
            // uint128 baseFeeX128 = 340282366920938463463374607431768211 45; // 10%
            // uint128 protocolTakeX128 = 340282366920938463463374607431768211 45;
↪   //10%

            uint256 amountA = startingTarget / 10 ** (18 - tokenA.decimals());
            uint256 amountB = startingTarget / 10 ** (18 - tokenB.decimals());

            tokenA.mint(owner, amountA);
            tokenA.approve(diamond, amountA);
            tokenB.mint(owner, amountB);
            tokenB.approve(diamond, amountB);
            SimplexFacet(diamond).addClosure(cid, startingTarget, baseFeeX128,
↪   protocolTakeX128);
        }

        vm.stopPrank();
    }

    function testAudit_CollectingEarningsEatsUpProtocolFee() public {
        uint16 cid = 3;

        // Alice adds value
        address alice = makeAddr("Alice");
        tokenA.mint(alice, 1000e18 + 1);
        tokenB.mint(alice, 1000e6 + 1);
```

```
        vm.startPrank(alice);
        tokenA.approve(diamond, 1000e18 + 1);
        tokenB.approve(diamond, 1000e6 + 1);
        ValueFacet(diamond).addValue(alice, cid, 2000e18, 0);
        vm.stopPrank();

        // Mock vault earnings
        {
            tokenA.mint(address(vaultA), 3e18);
        }

        // Mock disable vault withdrawal
        {
            vm.mockCall(
                address(vaultA),
                abi.encodeWithSelector(ERC4626.maxWithdraw.selector, diamond),
                abi.encode(0)
            );
        }

        // Mock protocol earnings in the contract
        {
            tokenA.mint(diamond, 6e18);
        }

        // There are protocol fees
        assertEq(tokenA.balanceOf(diamond), 6e18);

        // Alice collects earnings when the underlying vault is disabled
        vm.prank(alice);
        ValueFacet(diamond).collectEarnings(alice, cid);

        // The protocol fee is lost
        assertApproxEqAbs(tokenA.balanceOf(diamond), 0, 4e9);
    }
}

contract MockERC4626WithFees is ERC4626Fees {

    uint256 entryFeeBasisPoints;
    uint256 exitFeeBasisPoints;
    address feeRecipient;

    constructor(
        IERC20 asset_,
        string memory name_,
        string memory symbol_
    ) ERC4626(asset_) ERC20(name_, symbol_) {
        feeRecipient = msg.sender;
    }
```

```
    function setEntryFee(uint256 entryFeeBasisPoints_) public {
        entryFeeBasisPoints = entryFeeBasisPoints_;
    }

    function setExitFee(uint256 exitFeeBasisPoints_) public {
        exitFeeBasisPoints = exitFeeBasisPoints_;
    }

    function _entryFeeBasisPoints() internal view override returns (uint256) {
        return entryFeeBasisPoints;
    }

    function _exitFeeBasisPoints() internal view override returns (uint256) {
        return exitFeeBasisPoints;
    }

    function _entryFeeRecipient() internal view override returns (address) {
        return feeRecipient;
    }

    function _exitFeeRecipient() internal view override returns (address) {
        return feeRecipient;
    }
}
```

Mitigation

It is recommended to revert withdrawal transaction if the underlying vault is temporarily disabled.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/72

Issue M-4: The value of each closure is not the same, and the same ValueToken cannot be used for all cids

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/280

Found by

Drynooo

Summary

In ValueTokenFacet.sol, the mint/burn function uses the same ValueToken regardless of the cid.This can lead to an attacker being able to add liquidity to a low value closure to get a ValueToken, and then take out liquidity in a high value closure to get a more valuable more tokens.

In addition, the protocol hopes that when a token in the pool crashes unexpectedly, it will not affect users who have not added such liquidity. But it is not feasible under this kind of value design. Let's say user A adds liquidity for 15 tokens, but doesn't add aBTC. Then aBTC plummeted, which should not affect user A. However, users of other liquidity pools can exchange mint/burn ValueToken for the liquidity of the closure where user A is located, so as to avoid their losses. The loss is borne by users like user A.

Root Cause

In ValueTokenFacet.sol, the mint/burn function uses the same ValueToken regardless of the cid.

Internal Pre-conditions

1. In the pool where the attacker wants to take out the mobility, there needs to be another user who has already minted the ValueToken, otherwise the mobility will reach maxValue and revert when burning the ValueToken.

External Pre-conditions

none

Attack Path

1. There is a difference in the value of the liquidity of two pools. In normal market volatility, this must exist.
2. The attacker adds liquidity to the lower value pool cidA.
3. The attacker mint the ValueToken through cidA, and then burn the ValueToken to obtain the liquidity of cidB
4. Remove the liquidity of cidB.
5. The attacker gets more value.

Impact

Liquidity providers lose value. Attackers can steal liquidity without paying any cost.

PoC

Put the following poc under the test/facets folder run `forge test --mt testdiff_cid_valueToken -vvv`

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;

import {MultiSetupTest} from "./MultiSetup.u.sol";
import {console2 as console} from "forge-std/console2.sol";
import "../../src/multi/facets/ValueFacet.sol";
import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol";
import {AssetBookImpl} from "../../src/multi/Asset.sol";
import {MAX_TOKENS} from "../../src/multi/Token.sol";
import {MockERC20} from "../mocks/MockERC20.sol";
import {MockERC4626} from "../mocks/MockERC4626.sol";

contract ValueFacetTest is MultiSetupTest {
    function setUp() public {
        vm.startPrank(owner);
        _newDiamond();
        _newTokens(4);
        _fundAccount(alice);
        _fundAccount(bob);
        // Its annoying we have to fund first.
        _fundAccount(address(this));
        _fundAccount(owner);
        // So we have to redo the prank.
        vm.startPrank(owner);
        _initializeClosure(0xF, 100e18); // 1,2,3,4
        _initializeClosure(0xE, 100e18); // 2,3,4
        _initializeClosure(0xD, 100e18); // 1,3,4
```

```
        _initializeClosure(0xc, 1e12); // 3,4
        _initializeClosure(0xB, 100e18); // 1,2,4
        _initializeClosure(0xa, 1e12); // 2,4
        _initializeClosure(0x9, 1e18); // 1,4
        _initializeClosure(0x8, 1e12); // 4
        _initializeClosure(0x7, 100e18); // 1,2,3
        _initializeClosure(0x6, 100e18); // 2,3
        _initializeClosure(0x5, 1e28); // 1,3
        _initializeClosure(0x4, 1e12); // 3
        _initializeClosure(0x3, 100e18); // 1,2
        _initializeClosure(0x2, 1e12); // 2
        _initializeClosure(0x1, 1e12); // 1
        vm.stopPrank();
    }

    // Test whether the token values of different cids should be equal
    function testdiff_cid_valueToken() public {
        // Simulates market price changes, where the price of tokens[0] is smaller
↪   than that of tokens[1]
        (uint256 inAmount, uint256 outAmount) = swapFacet.swap(
            bob, // recipient
            tokens[0], // tokenIn
            tokens[1], // tokenOut
            int256(1e19), // positive for exact input
            0, // no price limit{}
            0x3
        );

        // Simulating market price changes, the price of tokens[1] is smaller than
↪   that of tokens[2]
        (uint256 inAmount1, uint256 outAmount1) = swapFacet.swap(
            bob, // recipient
            tokens[1], // tokenIn
            tokens[2], // tokenOut
            int256(1e19), // positive for exact input
            0, // no price limit{}
            0x6
        );
        // Last state, the price sorting in the market: token[0]<token[1]<token[2].
↪   This state must exist in reality.

        // bob is a normal user, and after adding liquidity, the normal mint
↪   valueToken
        valueFacet.addValue(bob, 0x6, 1e19, 0);
        vm.prank(bob);
        valueTokenFacet.mint(1e19, 0, 0x6);


        // alice, as the attacker, uses Single for ease of showing gains
        vm.startPrank(alice);
```

```
        uint256 requiredBalance = valueFacet.addValueSingle(
            alice,
            0x3,
            1e19,
            0,
            tokens[1],
            0
        );

        valueTokenFacet.mint(1e19, 0, 0x3);

        valueTokenFacet.burn(1e19, 0, 0x6);

        uint256 received = valueFacet.removeValueSingle(
            alice,
            0x6,
            1e19,
            0,
            tokens[1],
            0
        );
        vm.stopPrank();

        // If it's positive, the attack was successful. This is just to verify that
 ↪  this issue exists and that there must be a more profitable way to attack.
        console.log("diff:", int(received) - int(requiredBalance));
    }
}
```

Mitigation

It is recommended that different valueTokens should be used for different cids.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/80

Issue M-5: Invariant Breaking1

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/319

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xNov1ce, future

Summary

In the `addValue` function, both `Closure.balances` and `Closure.targetX128` increase proportionally without control by the `deMinimus`. Consequently, the delta between `the sum of values` and the `target` also increases, which could lead to exceeding the designed limits.

Root Cause

https://github.com/sherlock-audit/2025-04-burve/tree/main/Burve/src/multi/closure/Closure.sol#L127-L142

```
    function addValue(
        Closure storage self,
        uint256 value,
        uint256 bgtValue
    ) internal returns (uint256[MAX_TOKENS] memory requiredBalances) {
        trimAllBalances(self);
        // Round up so they add dust.
        uint256 scaleX128 = FullMath.mulDivX256(
            value,
            self.n * self.targetX128,
            true
        );
        uint256 valueX128 = value << 128;
        // Technically, by rounding up there will be a higher target value than
↪  actual value in the pool.
        // This is not an issue as it causes redeems to be less by dust and swaps
↪  to be more expensive by dust.
        // Plus this will be fixed when someone adds/removes value with an exact
↪  token amount.
127:    self.targetX128 +=
            valueX128 /
            self.n +
            ((valueX128 % self.n) > 0 ? 1 : 0);
        self.valueStaked += value;
        self.bgtValueStaked += bgtValue;
```

```
        // Value is handled. Now handle balances.
        for (uint8 i = 0; i < MAX_TOKENS; ++i) {
            if (!self.cid.contains(i)) continue;
            requiredBalances[i] = FullMath.mulX128(
                scaleX128,
                self.balances[i],
                true
            );
            // This happens after because the vault will have
142:        self.setBalance(i, self.balances[i] + requiredBalances[i]);
        }
    }
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

PoC

1. Add this function: https://github.com/sherlock-audit/2025-04-burve/tree/main/Burve/src/multi/closure/Closure.sol#L893

```
function realValue(Closure storage self) internal returns (uint256 _realValueX128) {
    uint256[MAX_TOKENS] storage esX128 = SimplexLib.getEsX128();
    for (uint8 i = 0; i < MAX_TOKENS; ++i) {
        if (!self.cid.contains(i)) continue;
        _realValueX128 += ValueLib.v(self.targetX128, esX128[i], self.balances[i],
↪   false);
    }
}
```

2. Change this function: https://github.com/sherlock-audit/2025-04-burve/tree/main/Burve/src/multi/facets/ValueFacet.sol#L280

```
function queryValue(address owner, uint16 closureId) external
    returns (
        uint256 valueX128,
```

```
        uint256 targetX128,
        uint256[MAX_TOKENS] memory,
        uint256 deMinimusX128
    ) {
    ClosureId cid = ClosureId.wrap(closureId);
    Closure storage c = Store.closure(cid); // Validates cid.
    valueX128 = c.realValue();
    targetX128 = c.targetX128;
    SearchParams memory search = Store.simplex().searchParams;
    deMinimusX128 = uint256(search.deMinimusX128);
}
```

3. Add this testing function: https://github.com/sherlock-audit/2025-04-burve/tree/main/Burve/test/facets/ValueFacet.t.sol

```
function test_invariant1() public {
    SearchParams memory sp = SearchParams(10, 500 << 128, 1e18);
    vm.prank(owner);
    simplexFacet.setSearchParams(sp);

    valueFacet.addSingleForValue(alice, 0x7, tokens[1], 0.52019107e20, 0, 0);

    uint256 valueX128;
    uint256 targetX128;
    uint256 deMinimusX128;

    console.log("Step1:");
    (valueX128, targetX128, , deMinimusX128) = valueFacet.queryValue(address(0),
↪   0x7);
    console.log("valueX128     :", valueX128);
    console.log("targetX128    :", targetX128);
    console.log("deMinimusX128 :", deMinimusX128);
    console.log("valueX128 - targetX128 * 3                  : ",
↪   (int256)(valueX128) - (int256)(targetX128) * 3);
    console.log("");

    valueFacet.addValue(alice, 0x7, 3e11, 0);
    valueFacet.addValue(alice, 0x7, 3e16, 0);
    valueFacet.addValue(alice, 0x7, 3e20, 0);
    valueFacet.addValue(alice, 0x7, 3e23, 0);

    console.log("Step2:");
    (valueX128, targetX128, , deMinimusX128) = valueFacet.queryValue(address(0),
↪   0x7);
    console.log("valueX128     :", valueX128);
    console.log("targetX128    :", targetX128);
    console.log("deMinimusX128 :", deMinimusX128);
    console.log("1<<128        : %e", uint256(1<<128));
    console.log("valueX128 - targetX128 * 3                  : %e",
↪   (int256)(valueX128) - (int256)(targetX128) * 3);
```

```
    console.log("valueX128 - (targetX128 + deMinimusX128) * 3 : %e",
↪   (int256)(valueX128) - (int256)(targetX128 + deMinimusX128) * 3);
}
```

forge test --match-test "test_invariant1" -vv If deMinimus = 500 « 128, Result:

```
Ran 1 test for test/facets/ValueFacet.t.sol:ValueFacetTest
[PASS] test_invariant() (gas: 1851203)
Logs:
  Step1:
  valueX128     : 11931494446589408476617528898861097455198561588475744725603 5
  targetX128    : 39771648155298028255391763002698103876158974815006309630937
  deMinimusX128 : 170141183460469231731687303715884105728000
  valueX128 - targetX128 * 3                    :  -194833370764913085602614816367 76

  Step2:
  valueX128     : 10230611993939680700250295445985513553608552586928280956236878 6
  targetX128    : 34102039979798935667015918447015107849047050355960606309630937
  deMinimusX128 : 170141183460469231731687303715884105728000
  1<<128        : 3.402823669209384634633746074317682114 56e38
  valueX128 - targetX128 * 3                    :
↪   1.455199118809811988944374801400990633475975e42
  valueX128 - (targetX128 + deMinimusX128) * 3 :
↪   9.447755684284042937493128902533383162919 75e41
```

If deMinimus = 500, Result:

```
Ran 1 test for test/facets/ValueFacet.t.sol:ValueFacetTest
[PASS] test_invariant() (gas: 1860320)
Logs:
  Step1:
  valueX128     : 11931494446589408476617528898854177250294406775327144791511 6
  targetX128    : 39771648155298028255391762996180590834314689251090482638472
  deMinimusX128 : 500
  valueX128 - targetX128 * 3                    :  -300

  Step2:
  valueX128     : 10230611993939680700250295445985506633403648432115127611647 7882
  targetX128    : 34102039979798935667015918447008590336005206070396690482638472
  deMinimusX128 : 500
  1<<128        : 3.402823669209384634633746074317682114 56e38
  valueX128 - targetX128 * 3                    :
↪   1.455199118829295326020866109961204668562466e42
  valueX128 - (targetX128 + deMinimusX128) * 3 :
↪   1.455199118829295326020866109961204668560966e42
```

Impact

In the ReadMe:

> The ”value” of the closure (according to the formulas) can never be more than deMinimus * (number of tokens in the closure) from the target value of the pool times number of tokens in the closure.

1. Invariant Breaking.
2. This could result in unfairness for users.

Mitigation

Consider recalculating the `target` using `ValueLib.t`.

Issue M-6: Incorrect earnings calculation in removeValueSingle() function causes partial user losses

Source: https://github.com/sherlock-audit/2025-04-burve-judging/issues/422

Found by

Drynooo, TessKimy, bladeee, future, h2134, nganhg, rsam_eth, zark

Summary

Asset removal before invoking `trimBalance()` in `removeValueSingle()` function causes partial loss of user earnings, as the asset's `remove()` function uses outdated earnings per value variable to calculate user collected earnings.

Root Cause

In ValueFacet.sol:229 the asset's `remove()` function is invoked prior to the closure's `removeValueSingle()`. (resulting in outdated earnings per value calculations) In Asset.sol:114 the asset's `remove()` function invoke `collect` function. In Asset.sol:143 the asset's `collect()` function gets outdated earnings per value from the closure.

Internal Pre-conditions

1. Closures deposit their fee earnings into an active vault.

External Pre-conditions

1. Vault distributes reward to closures.

Attack Path

1. User call `removeValueSingle()` function.

Impact

Users may experience a loss exceeding 1% of their earnings if the pool is not sufficiently active, causing the `earnings per value` to remain outdated for an extended period.

PoC

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.27;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";
import {SimplexDiamond as BurveDiamond} from "../../src/multi/Diamond.sol";
import {SimplexFacet} from "../../src/multi/facets/SimplexFacet.sol";
import {SwapFacet} from "../../src/multi/facets/SwapFacet.sol";
import {ValueFacet} from "../../src/multi/facets/ValueFacet.sol";
import {VaultFacet} from "../../src/multi/facets/VaultFacet.sol";
import {MockERC20} from "../mocks/MockERC20.sol";

contract PoC1 is Test {
    address payable diamond;
    BurveDiamond public burveDiamond;
    SimplexFacet public simplexFacet;
    SwapFacet public swapFacet;
    ValueFacet public valueFacet;
    VaultFacet public vaultFacet;
    uint16 cid = 3;

    address user = address(1);
    address randomUser = address(2);
    address fakeRewardDistributor = address(3);
    address[] tokens;
    address[] vaults;

    uint128 addValue = 10_000;
    uint128 bgtValue = 1_000;
    uint256 tokenAmount = 1000 * 1e18;

    function setUp() public {
        // address received after run script/Deploy.s.sol
        diamond = payable(0xa513E6E4b8f2a923D98304ec87F64353C4D5C853);
        burveDiamond = BurveDiamond(diamond);
        simplexFacet = SimplexFacet(diamond);
        swapFacet = SwapFacet(diamond);
        valueFacet = ValueFacet(diamond);
        vaultFacet = VaultFacet(diamond);

        // get tokens list
        tokens = simplexFacet.getTokens();

        // get vaults list
        vaults = new address[](tokens.length);
        for (uint256 i = 0; i < tokens.length; i++) {
            (vaults[i], ) = vaultFacet.viewVaults(tokens[i]);
```

```
        }

        vm.deal(user, 1000 * 1e18);
        vm.deal(randomUser, 1000 * 1e18);
        for (uint256 i = 0; i < tokens.length; i++) {
            deal(tokens[i], user, type(uint256).max);
            deal(tokens[i], randomUser, type(uint256).max);
            deal(tokens[i], fakeRewardDistributor, type(uint256).max);
            vm.startPrank(user);
            MockERC20(tokens[i]).approve(diamond, type(uint256).max);
            vm.stopPrank();
            vm.startPrank(randomUser);
            MockERC20(tokens[i]).approve(diamond, type(uint256).max);
            vm.stopPrank();
        }

        // User adds liquidity
        vm.startPrank(user);
        valueFacet.addValue(user, cid, addValue, bgtValue);
        vm.stopPrank();

        // fake vault reward distribution
        vm.startPrank(fakeRewardDistributor);
        for (uint256 i = 0; i < tokens.length; i++) {
            MockERC20(tokens[i]).transfer(vaults[i], tokenAmount);
        }
        vm.stopPrank();
    }

    function testNormalRemoveValueSingle() public {
        // User removes value
        vm.startPrank(user);
        valueFacet.removeValueSingle(user, cid, addValue, bgtValue, tokens[0], 0);

        // User collects earnings
        (uint256[16] memory collectedBalance, uint256 collectedBgt) = valueFacet
            .collectEarnings(user, cid);
        vm.stopPrank();

        for (uint256 i = 0; i < tokens.length; i++) {
            console2.log(
                "Collected balance of token %s: %s",
                tokens[i],
                collectedBalance[i]
            );
        }
    }

    function testRemoveValueSingleAfterCallTrimBalances() public {
        // Random user adds value => triggers trimBalances()
```

```
        vm.startPrank(randomUser);
        valueFacet.addValue(randomUser, cid, addValue, bgtValue);
        vm.stopPrank();

        // User removes value
        vm.startPrank(user);
        valueFacet.removeValueSingle(user, cid, addValue, bgtValue, tokens[0], 0);

        // User collects earnings
        (uint256[16] memory collectedBalance, uint256 collectedBgt) = valueFacet
            .collectEarnings(user, cid);
        vm.stopPrank();

        for (uint256 i = 0; i < tokens.length; i++) {
            console2.log(
                "Collected balance of token %s: %s",
                tokens[i],
                collectedBalance[i]
            );
        }
    }
}
```

Mitigation

Move the asset's `remove()` function below the closure's `removeValueSingle()`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/itos-finance/Burve/pull/69

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.