Burve Security Review

Pashov Audit Group

Conducted by: mahdiRostami, Stormy, samuraii77, btk

January 29th 2025 - February 6th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **itos-finance/Burve** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Burve

Burve is a multi-pool system that market-makes up to 16 tokens using a curve-like stable swap built from Uniswap positions. It uses "implied" AMMs, ERC4626 vaults, and user-managed Closures to manage liquidity and enable trading across 120 token pairs with just 16 token deposits.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - f0597768fee2d00c17429941f4721475e1ca5723

fixes review commit hash - e89ebff2c7daafc98e94c66e4273e4c366949c76

Scope

The following smart contracts were in scope of the audit:

- `Burve`
- `KodiakIsland`
- `TransferHelper`
- `EdgeFacet`
- `LiqFacet`
- `SimplexFacet`
- `SwapFacet`
- `Asset`
- `Closure`
- `Diamond`
- `E4626`
- `Edge`
- `LPToken`
- `Store`
- `BurveDeployLib.sol`
- `Token`
- `UniV3Edge`
- `VaultProxy`
- `Vertex`

7. Executive Summary

Over the course of the security review, mahdiRostami, Stormy, samuraii77, btk engaged with Itos Finance to review Burve. In this period of time a total of **20** issues were uncovered.

Protocol Summary

Protocol Name	Burve
Repository	https://github.com/itos-finance/Burve
Date	January 29th 2025 - February 6th 2025
Protocol Type	AMM

Findings Count

Severity	**Amount**
Critical	4
High	3
Medium	5
Low	8
Total Findings	**20**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect parenthesis usage results in a completely wrong calculation	Critical	Resolved
[C-02]	Incorrect cumulativeValue in LiqFacet::addLiq gives less shares to users	Critical	Resolved
[C-03]	Unrestricted diamondCut allows unauthorized facet modifications	Critical	Resolved
[C-04]	Draining approved tokens by unrestricted uniswapV3MintCallback()	Critical	Resolved
[H-01]	Missing transfer of tokens before island.mint()	High	Resolved
[H-02]	Normalizing token weights can result in a revert	High	Resolved
[H-03]	Incorrect recipient results in the inability to properly redeem a position	High	Resolved
[M-01]	No slippage protection	Medium	Resolved
[M-02]	LiqFacet::removeLiq temporarily disabled when vault is paused	Medium	Acknowledged
[M-03]	Fee-on-transfer token compatibility Issue in protocol Burve::mint	Medium	Acknowledged
[M-04]	Missing selectors from SimplexFacet for diamond cuts	Medium	Resolved
[M-05]	Missing supportsInterface Selector from DiamondLoupeFacet	Medium	Resolved

[L-01]	LPToken.burn() spend allowances even when account == msg.sender	Low	Resolved
[L-02]	ERC20 implementations may revert on zero approval	Low	Resolved
[L-03]	Calculating total assets on deposit misses temp withdrawal subtraction	Low	Resolved
[L-04]	Tokens such as USDT will be unusable	Low	Resolved
[L-05]	Some pools will be impossible to incorporate	Low	Acknowledged
[L-06]	Unregistered token mistaken for token 0 in Closure::newClosureId	Low	Resolved
[L-07]	Receive function implemented but no method to withdraw ETH	Low	Acknowledged
[L-08]	Incorrect boundary check for lowTick and highTick	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect parenthesis usage results in a completely wrong calculation

Severity

Impact: High

Likelihood: High

Description

Upon calling `Edge.calcLowerImplied()` when the price is below the narrow range low tick, we have the following piece of code:

```
uint256 xyX192 = (x << (128 / y)) << 64;
```

It aims to compute the `x/y` ratio in 192-bit precision by first moving `x` to 128-bit precision, dividing by `y` and then going to 192-bit precision when shifting left by 64 bits. However, the formula has a critical issue as `x` is shifted left by `128 / y` bits due to the parenthesis usage, this will round down to 0 in pretty much every case which results in the variable simply holding the value of `x` shifted left by 64 bits.

This vulnerability is also in a few other functions.

Recommendations

```
+ uint256 xyX192 = ((x << 128) / y) << 64;
- uint256 xyX192 = (x << (128 / y)) << 64;
```

[C-02] Incorrect `cumulativeValue` in `LiqFacet::addLiq` gives less shares to users

Severity

Impact: High

Likelihood: High

Description

Upon adding liquidity, we compute the shares to give to the user using the following code:

```
uint256 addedBalance = tokenBalance - preBalance[idx];

uint256 cumulativeValue = tokenBalance;

        TokenRegistry storage tokenReg = Store.tokenRegistry();

        for (uint256 i = 0; i < n; ++i) {

            if (i == idx) {

                continue;

            } else if (preBalance[i] != 0) {

                address otherToken = tokenReg.tokens[i];

                Edge storage e = Store.edge(token, otherToken);

                uint256 priceX128 = (token < otherToken) ? e.getInvPriceX128(
                  token<otherToken
                ) ? e.getInvPriceX128
                  (tokenBalance, preBalance[i]

                cumulativeValue += FullMath.mulX128
                  (preBalance[i], priceX128, true);

            }

        }

        shares = AssetLib.add(recipient, cid, addedBalance, cumulativeValue);
```

It computes the total value of all tokens in terms of the token we are adding liquidity in and store that in `cumulativeValue`. We then use the following formula for the shares where `num` is the `addedBalance` inpu, `denom` is the `cumulativeValue` input and `total` is the total amount of shares:

```
shares = FullMath.mulDiv(num, total, denom);
```

It simply downscale the total amount of shares based on the provided value by the user and the total value. The issue is that `cumulativeValue` includes the deposit by the user which results in him receiving less shares as the total value is bigger.

Let's imagine the following scenario:

1. The `preBalance` of the token deposited is 10 and the added balance is also 10, thus `tokenBalance` is 20
2. We wrongly set `cumulativeValue` to `tokenBalance` or 20
3. We imagine that cumulative value is increased twice by 10 in the loop, to a total of 40
4. The shares for the user will be `10 * 100 / 40 = 25` (assuming total shares are 100), the total shares go to 125 and the user shares are 25
5. If a user withdraws, he will be able to get `25 / 125 = 0.2` of the total value which we computed to be 40, thus 8 (2 less than what he deposited, he should get the same)

This is because `cumulativeValue` should be initialized with the value of `preBalance[idx]`, that way the formula for the user shares will be `10 * 100 / 30 = 0.33` (30 comes from 10 tokens from the `preBalance[idx]` and 2 * 10 from the other tokens). Now, when the user withdraws, he will be able to get `33 / 133 = 0,2481203008` of the 40 assets which is ~10.

Recommendations

Modify `LiqFacet::addLiq` as follows:

```
-       uint256 cumulativeValue = tokenBalance;
+       uint256 cumulativeValue = preBalance[idx];
```

Remove the following check in `AssetLib::add`:

```
require(num == denom, "NDE");
```

[C-03] Unrestricted `diamondCut` allows unauthorized facet modifications

Severity

Impact: High

Likelihood: High

Description

Description

The `SimplexDiamond` contract includes `DiamondCutFacet.diamondCut.selector` in its selectors. This function allows adding, removing, or modifying facet cuts, which determine the contract's functionality. However, **this function is not restricted**, meaning **anyone** can call it to remove or replace any selector or facet.

Proof of Concept (PoC)

The test case below demonstrates how a random user can call `diamondCut` to remove or modify contract functionality, potentially leading to loss of control over the contract.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.17;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";
import {BurveDeploymentLib} from "../../src/deployment/BurveDeployLib.sol";
import {SimplexDiamond} from "../../src/multi/Diamond.sol";
import {StorageFacet} from "../mocks/StorageFacet.sol";
import {IDiamond} from "Commons/Diamond/interfaces/IDiamond.sol";
import {LibDiamond} from "Commons/Diamond/libraries/LibDiamond.sol";

contract EdgeFacetTest is Test {
    SimplexDiamond public diamond;
    StorageFacet public storageFacet;
    address public owner = makeAddr("owner");
    StorageFacet public storageFacetContract;

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

        // Deploy the diamond and facets
        (
          addressliqFacetAddr,
          addresssimplexFacetAddr,
          addressswapFacetAddr
        ) = BurveDeploymentLib.deployFacets(

        // Deploy storage facet
        storageFacetContract = new StorageFacet();

        // Create the diamond with initial facets
        diamond = new SimplexDiamond
          (liqFacetAddr, simplexFacetAddr, swapFacetAddr);

        // Add storage facet using LibDiamond directly since we're the owner
        bytes4[] memory selectors = new bytes4[](1);
        selectors[0] = StorageFacet.getEdge.selector;

        IDiamond.FacetCut[] memory cuts = new IDiamond.FacetCut[](1);
        cuts[0] = IDiamond.FacetCut({
            facetAddress: address(storageFacetContract),
            action: IDiamond.FacetCutAction.Add,
            functionSelectors: selectors
        });

        LibDiamond.diamondCut(cuts, address(0), "");

        vm.stopPrank();
    }

    function test_rediamondCut() public {
        vm.startPrank(makeAddr("random_user"));
        // Add storage facet using LibDiamond directly since we're the owner
        bytes4[] memory selectors = new bytes4[](1);
        selectors[0] = StorageFacet.getEdge.selector;

        IDiamond.FacetCut[] memory cuts = new IDiamond.FacetCut[](1);
        cuts[0] = IDiamond.FacetCut({
            facetAddress: address(0),
            action: IDiamond.FacetCutAction.Remove,
            functionSelectors: selectors
        });

        LibDiamond.diamondCut(cuts, address(0), "");
    }
}
```

Recommendations

Restrict the `diamondCut` function to only be callable by an authorized admin by implementing `AdminLib.validateOwner()` or similar permission checks.

[C-04] Draining approved tokens by unrestricted `uniswapV3MintCallback()`

Severity

Impact: High

Likelihood: High

Description

`Burve::uniswapV3MintCallback` is an external function with no access control. This function takes three parameters: `amount0Owed`, `amount1Owed`, and `data`. It then decodes `data` to get an address and transfers tokens from that address to the liquidity pool (lp). An attacker could see which addresses have approved to `Burve.sol` and transfer tokens from those addresses to the pool.

```
function uniswapV3MintCallback
    (uint256 amount0Owed, uint256 amount1Owed, bytes calldata data) external {
        address source = abi.decode(data, (address));
        TransferHelper.safeTransferFrom(token0, source, address
          (pool), amount0Owed);
        TransferHelper.safeTransferFrom(token1, source, address
          (pool), amount1Owed);
    }
```

1. Provide `data` which decodes to a user address with a hanging approval to `Burve` and amounts equal to the approved amounts
2. As the function has no access control, the funds will be transferred into the pool
3. This causes a direct loss of funds for the users

Recommendations

Restrict this function so that only the pool can call it.

8.2. High Findings

[H-01] Missing transfer of tokens before island.mint()

Severity

Impact: Medium

Likelihood: High

Description

The Burve contract allows users to provide liquidity to both the island pool and Uniswap V3 pools. When adding liquidity to the island pool, it will transfer tokens in from the sender (which is the Burve contract itself) before providing liquidity to Uniswap.

```
function mint(address recipient, uint128 liq) external {
        for (uint256 i = 0; i < distX96.length; ++i) {
            uint128 liqAmount = uint128(shift96(liq * distX96[i], true));
            mintRange(ranges[i], recipient, liqAmount);
        }

        _mint(recipient, liq);
    }
```

However, when users call `Burve.mint()`, it does not transfer the necessary tokens beforehand. This results in a failed transaction, preventing users from successfully providing liquidity to the island pool through the Burve contract.

Recommendations

Ensure the required tokens are transferred and approved before calling `island.mint()`.

[H-02] Normalizing token weights can result in a revert

Severity

Impact: High

Likelihood: Medium

Description

Upon normalizing weights when having a swap, we have this piece of code to turn the `weights` array from an array with amounts to an array with percentages:

```
self.weights[i] = FullMath.mulDivX256(
                self.weights[i],
                self.totalWeight
            );
```

The issue is that if there is a single element in the array, then that element will equal the `totalWeight` which results in a revert when calling `mulDivX256()` as we have the following check there:

```
require(denominator > num, "0");
```

As they are equal, we will revert.

Recommendations

When the values are equal, set the `weights` element to `2 ^ 256 - 1` or `type(uint256).max`

[H-03] Incorrect recipient results in the inability to properly redeem a position

Severity

Impact: Medium

Likelihood: High

Description

Upon minting, we have the following code:

```
if (range.lower == 0 && range.upper == 0) {
            uint256 mintShares = islandLiqToShares(liq);
            island.mint(mintShares, recipient);
        } else {
            // mint the V3 ranges
            pool.mint(address(this), range.lower, range.upper, liq, abi.encode
              (msg.sender));
        }
```

When minting for the island, we set the recipient as the `recipient` input and when minting Uniswap V3 liquidity, we set the recipient as `address(this)`. The latter is correct while the former is not. This is because when burning, the shares will be burned from the caller of the `burn()` function on the target, which will be the `Burve` contract. As the `Burve` contract does not have the minted shares when minting for the island, we will simply revert.

The user still has 2 options, thus the medium impact:

- batch a transaction by transferring the shares to the `Burve` contract and then burning the liquidity, this will result in the correct result
- simply burn his shares directly on the island, note that this will result in the user still having the minted `Burve` shares

Recommendations

```
-  island.mint(mintShares, recipient);
+  island.mint(mintShares, address(this));
```

8.3. Medium Findings

[M-01] No slippage protection

Severity

Impact: Medium

Likelihood: Medium

Description

The Burve contract allows users to provide liquidity to both the Island pool and Uniswap V3 pools. The number of shares minted to users is determined using the following calculation:

```
function islandLiqToShares(uint128 liq) internal view returns
      (uint256 shares) {
        if (address(island) == address(0x0)) {
            revert NoIsland();
        }

        (uint160 sqrtRatioX96,,,,,,) = pool.slot0();

        (uint256 amount0, uint256 amount1) = getAmountsFromLiquidity(
            sqrtRatioX96,
            island.lowerTick(),
            island.upperTick(),
            liq
        );

        (,, shares) = island.getMintAmounts(amount0, amount1);
    }
```

Since the amount of shares depends on `sqrtRatioX96`, any fluctuations in this value will impact the number of shares users receive. As a result, users may end up with fewer tokens than expected.

Recommendations

Consider adding a `minSharesOut` parameter to the mint/burn functions.

[M-02] `LiqFacet::removeLiq` temporarily disabled when vault is paused

Severity

Impact: Medium

Likelihood: Medium

Description

In `LiqFacet`, when users add liquidity, they deposit a single token into a specific vault:

```
function addLiq(
        address recipient,
        uint16 _closureId,
        address token,
        uint128 amount
    ) external nonReentrant returns (uint256 shares) {
```

However, when removing liquidity, the function attempts to withdraw from all vaults where the token is in the closure:

```
for (uint8 i = 0; i < n; ++i) {
            VertexId v = newVertexId(i);
            VaultPointer memory vPtr = VaultLib.get(v);
            uint128 bal = vPtr.balance(cid, false);
            if (bal == 0) continue;
            // If there are tokens, we withdraw.
            uint256 withdraw = FullMath.mulX256(percentX256, bal, false);
            vPtr.withdraw(cid, withdraw);
            vPtr.commit();
            address token = tokenReg.tokens[i];
            TransferHelper.safeTransfer(token, recipient, withdraw);
        }
```

The issue arises when one of these vaults is paused. Since the function does not account for paused vaults, the entire withdrawal process is blocked, preventing users from removing liquidity.

Recommendations

Implement a mechanism to skip paused vaults and allow withdrawals from the remaining active vaults to ensure a smooth liquidity removal.

[M-03] Fee-on-transfer token compatibility Issue in protocol `Burve::mint`

Severity

Impact: Medium

Likelihood: Medium

Description

The `Burve::mint` function interacts with the `uniswapV3MintCallback`, which transfers `token0` and `token1` to the Uniswap pool. However, **fee-on-transfer** tokens (tokens that deduct a percentage as a fee on each transfer) are not properly accounted for in this implementation.

When using fee-on-transfer tokens, the amount received by the pool **will be less than the expected amount**, causing the minting process to fail due to an arithmetic underflow.

Proof of Concept (PoC)

To simulate the issue, I modified the `Burve::uniswapV3MintCallback` function by subtracting a fixed amount from the transferred values to mimic a fee deduction:

```
function uniswapV3MintCallback
  (uint256 amount0Owed, uint256 amount1Owed, bytes calldata data) external {
    address source = abi.decode(data, (address));
    TransferHelper.safeTransferFrom(token0, source, address
      (pool), amount0Owed - 10);
    TransferHelper.safeTransferFrom(token1, source, address
      (pool), amount1Owed - 10);
}
```

Test Case & Logs

Test:

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

import {Test, console} from "forge-std/Test.sol";
import {Burve, TickRange} from "../src/Burve.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract BurveTest is Test {
    Burve public burve;
    address public pool = 0xc473e2aEE3441BF9240Be85eb122aBB059A3B57c;
    address public USDC = 0xaf88d065e77c8cC2239327C5EDb3A432268e5831;
    address public holderu = 0xf89d7b9c864f589bbF53a82105107622B35EaA40;
    address public WETH = 0x82aF49447D8a07e3bd95BD0d56f35241523fBab1;
    address public holderw = 0x70d95587d40A2caf56bd97485aB3Eec10Bee6336;
    uint128[] public weights = [4249, 3893, 9122];
    TickRange public range = TickRange(-199800, -194880);
    TickRange[] public ranges = [range, range, range];
    address public alice = makeAddr("random_user");

    function setUp() public {
        // Create main net fork
        vm.createSelectFork
          ({urlOrAlias: "arbitrum_api", blockNumber: 260000000});
        burve = new Burve(pool, address(0), ranges, weights);
        vm.prank(holderu);
        IERC20(USDC).transfer(alice, 1e11);
        vm.prank(holderw);
        IERC20(WETH).transfer(alice, 1e20);
    }

    function test_fee() public {
        vm.startPrank(alice);
        IERC20(USDC).approve(address(burve), 1e11);
        IERC20(WETH).approve(address(burve), 1e20);
        burve.mint(alice, 100);
    }
}
```

Logs:

```
|   |   |   └─ ← [Return] 81792443700 [8.179e10]
|   |   |   ├─ [20564] Burve::uniswapV3MintCallback(
      90917[9.091e4],
      1,
      0x000000000000000000000000e2f70c5cbd298fb122db24264d8923e348cedae3
    )


    |   |   |   |   |   ├─ emit Transfer(
      from:random_user:[0xE2F70c5cbD298Fb122db24264d8923E348CeDaE3],
      to:0xc473e2aEE3441BF9240Be85eb122aBB059A3B57c,
      value:90907[9.09e4]
    )
    |   |   |   |   |   ├─ emit Approval(
      owner:random_user:[0xE2F70c5cbD298Fb122db24264d8923E348CeDaE3],
      spender:Burve:[0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f],
      value:9999999999999909093[9.999e19]
    )
    |   |   |   |   |   └─ ← [Return] true
    |   |   |   |   └─ ← [Return] true
    |   |   |   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
    |   |   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
    |   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
```

notes: same applies for multi token pool as well

[M-04] Missing selectors from `SimplexFacet` for diamond cuts

Severity

Impact: Medium

Likelihood: Medium

Description

Several functions from `SimplexFacet` are missing from the diamond cuts:

- `withdrawFees`
- `setName`
- `getName`

Since these functions are not included in the selector list, they cannot be called through the diamond contract, **effectively making them inaccessible**.

Recommendations

Ensure these functions are added as selectors in the diamond cut process.

```
{
-            bytes4[] memory simplexSelectors = new bytes4[](2);
+            bytes4[] memory simplexSelectors = new bytes4[](5);
             simplexSelectors[0] = SimplexFacet.addVertex.selector;
             simplexSelectors[1] = SimplexFacet.setDefaultEdge.selector;
+            simplexSelectors[2] = SimplexFacet.withdrawFees.selector;
+            simplexSelectors[3] = SimplexFacet.setName.selector;
+            simplexSelectors[4] = SimplexFacet.setDefaultEdge.getName;
             cuts[5] =
                 FacetCut(
                   {facetAddress:simplexFacet,
                   action:FacetCutAction.Add,
                   functionSelectors:simplexSelectors}
                 );
        }
```

[M-05] Missing `supportsInterface` Selector from `DiamondLoupeFacet`

Severity

Impact: Medium

Likelihood: Medium

Description

The `SimplexDiamond` contract is intended to support multiple interfaces, including `IERC165`, `IDiamondCut`, `IDiamondLoupe`, and `IERC173`. While `SimplexDiamond` correctly adds these interfaces to `ds.supportedInterfaces`, it **fails to include the `supportsInterface` function from `DiamondLoupeFacet` as a selector**. This omission **prevents the contract from properly supporting interfaces**, making it **incompatible with standard interface detection mechanisms**.

Recommendations

Ensure `supportsInterface` is included in the `DiamondLoupeFacet` selectors:

```
{
-            bytes4[] memory loupeFacetSelectors = new bytes4[](4);
+            bytes4[] memory loupeFacetSelectors = new bytes4[](5);
             loupeFacetSelectors[0] = DiamondLoupeFacet.facets.selector;

                         loupeFacetSelectors[1] = DiamondLoupeFacet.facetFunctionSelec
             loupeFacetSelectors[2] = DiamondLoupeFacet.facetAddresses.selector;
             loupeFacetSelectors[3] = DiamondLoupeFacet.facetAddress.selector;
+
+              loupeFacetSelectors[4] = DiamondLoupeFacet.supportsInterface.selector;
             cuts[1] = FacetCut({
                 facetAddress: address(new DiamondLoupeFacet()),
                 action: FacetCutAction.Add,
                 functionSelectors: loupeFacetSelectors
             });
```

8.4. Low Findings

[L-01] `LPToken.burn()` spend allowances even when `account == msg.sender`

The `LPToken.burn()` function burns a specified amount of LP tokens and redeems the underlying assets:

```
function burn(address account, uint256 shares) external {
        _spendAllowance(account, _msgSender(), shares);
        burveMulti.removeLiq(_msgSender(), ClosureId.unwrap(cid), shares);
        _burn(account, shares);
    }
```

Currently, the function deducts the allowance from `account` to `msg.sender` even when `account == msg.sender`. This means users must approve themselves before redeeming their tokens, leading to unnecessary gas costs and an inefficient user experience.

Consider skipping the allowance check when `account == msg.sender`:

```
if (account != _msgSender()) {
    _spendAllowance(account, _msgSender(), shares);
}
```

[L-02] ERC20 implementations may revert on zero approval

In the ERC4626 implementation, token allowances follow a pattern where approval is granted for the required amount before executing an operation and then reset to zero afterward:

```
self.token.approve(address(self.vault), assetsToDeposit);
            self.totalVaultShares += self.vault.deposit(
                assetsToDeposit,
                address(this)
            );
            self.token.approve(address(self.vault), 0);
```

However, some ERC20 implementations revert when approving a zero value. A notable example is BNB, which throws an error when approve(0) is called. This behavior can lead to a denial of service, preventing the protocol from functioning when such tokens are used. Consider using OpenZeppelin's forceApprove() instead.

[L-03] Calculating total assets on deposit misses temp withdrawal subtraction

Upon depositing in a vault, we compute the `totalAssets` like this:

```
uint256 totalAssets = temp.vars[0] + newlyAdding;
```

The issue is that it does not subtract `temp.vars[2]` which corresponds to the temp withdrawal amount. While this has no impact currently as there is no way for there to be a pending withdrawal amount during the deposit, any future changes to the code could result in this being a serious issue.

[L-04] Tokens such as USDT will be unusable

Upon committing after a swap and depositing into a vault, we have this piece of code:

```
self.token.approve(address(self.vault), assetsToDeposit);
self.totalVaultShares += self.vault.deposit(assetsToDeposit, address(this));
```

We are calling `approve()` on the token using the `IERC20` interface which expects a boolean return value. Tokens such as USDT do not return a boolean which will result in the call to always revert. Instead, consider using `forceApprove()` from the `SafeERC20` library.

[L-05] Some pools will be impossible to incorporate

Upon deploying the `Burve` contract, we call `Burve.nameFromPool()` which has the following piece of code:

```
name = string.concat(ERC20(t0).name(), "-", ERC20(t1).name
  (), "-Stable-KodiakLP");
```

It creates a name based on the concatenation of the `token0`'s and `token1`'s name. However, this incorrectly assumes that all tokens have a string symbol which is not the case - tokens such as `MKR` have a name and symbol of type `bytes32`. Instead, consider implementing a low-level call.

[L-06] Unregistered token mistaken for token 0 in `Closure::newClosureId`

The function `Closure::newClosureId` does not check whether a token is **registered** before using its index. If a token is **not registered**, the expression:

```
tokenReg.tokenIdx[tokens[i]]
```

will return `0`. The contract **mistakenly assumes** that the token index is valid and includes it in the `cid` (Closure ID).
If a user supplies an **unregistered token**, it is incorrectly mapped to **token 0** (the first registered token).

[L-07] Receive function implemented but no method to withdraw ETH

The contract includes a `receive` function, but there is no method to withdraw ETH from it.

[L-08] Incorrect boundary check for lowTick and highTick

The Edge.setRange() function allows the owner to initialize the lowTick and highTick parameters:

```
function setRange(
        Edge storage self,
        uint128 amplitude,
        int24 lowTick,
        int24 highTick
    ) internal {
```

Before assigning these values, the function enforces boundary checks:

```
self.lowTick = lowTick;
        require(lowTick > MIN_NARROW_TICK, "ERL");
        self.highTick = highTick;
        require(highTick < MAX_NARROW_TICK, "ERH");
```

These checks ensure that:

- `lowTick` is strictly greater than `MIN_NARROW_TICK`
- `highTick` is strictly less than `MAX_NARROW_TICK`

However, If `MIN_NARROW_TICK` and `MAX_NARROW_TICK` represent the absolute lowest and highest valid tick values, then the conditions are incorrect because:

- Using > instead of >= would unnecessarily exclude `MIN_NARROW_TICK`.
- Using < instead of <= would unnecessarily exclude `MAX_NARROW_TICK`.

Update the checks as follows:

```
require(lowTick >= MIN_NARROW_TICK, "ERL");
        require(highTick <= MAX_NARROW_TICK, "ERH");
```