Burve Security Review

Pashov Audit Group

Conducted by: eeyore, Shaka, saksham, Udsen

March 5th 2025 - March 13th 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 - b48765aba6f0d0172f5a75ad209ea5e2d4e352a3

fixes review commit hash - 61419ff42e43ee945e22a9ac3e1b743248b306b3

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, eeyore, Shaka, saksham, Udsen engaged with Itos Finance to review Burve. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Burve
Repository	https://github.com/itos-finance/Burve
Date	March 5th 2025 - March 13th 2025
Protocol Type	AMM

Findings Count

Severity	**Amount**
High	3
Medium	3
Low	6
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[H-01]	DoS on mint() and burn() due to overestimation of available liquidity	High	Resolved
[H-02]	First deposit front-running attack	High	Resolved
[H-03]	Island shares ownership not updated on Burve token transfer	High	Resolved
[M-01]	EnsureClosure would not add a vertex's closure if added in future	Medium	Resolved
[M-02]	FeeLib calculations can revert	Medium	Resolved
[M-03]	setAdjustor selector not included from SimplexFacet	Medium	Resolved
[L-01]	No check for duplicated island range	Low	Resolved
[L-02]	Importing interfaces from the forge-std/interfaces for production is not recommended	Low	Resolved
[L-03]	Missing sanity check for defaultEdge correctness	Low	Resolved
[L-04]	Inclusion of test-only ViewFacet facet	Low	Resolved
[L-05]	RemoveLiq does not check if the vertex is locked	Low	Acknowledged
[L-06]	Kodiac Island pause mechanism can lock funds	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] DoS on `mint()` and `burn()` due to overestimation of available liquidity

Severity

Impact: Medium

Likelihood: High

Description

In the Burve contract, `compoundV3Ranges()` is executed when `mint()` or `burn()` are called. This function performs the following actions:

1. Collects fees from the UniswapV3 pool for all ranges.
2. Calculates the total liquidity that can be minted with the available balance of token0 and token1.
3. Calculates the liquidity that can be minted for each range, according to their weight distribution.
4. Mints the respective liquidity for each range.

Step 2) is performed by the `collectAndCalcCompound()` function and these are the steps followed:

1. Obtains the balance of token0 and token1 in the Burve contract.
2. Calculates the amounts of token0 and token1 required to mint `2^64` liquidity given the weight distribution of the ranges.
3. Calculates the nominal liquidity for each token by scaling up by `2^64` the amounts of token0 and token1 and dividing by the amounts calculated in the previous step.

4. Caps the nominal liquidity to the minimum of the liquidity for token0 and token1, and adjusts for rounding errors in minting.

However, it is not taken into account that the distribution of the scaled distribution might not match the distribution of the actual liquidity in the pool, which overestimates the amount of liquidity that can be minted, causing the transaction to fail.

Let's consider the following example:

- The Burve contract has two ranges with equal weight distribution.
- On burn, the balance of token0 and token1 is 1.
- `amount0InUnitLiqX64` and `amount1InUnitLiqX64` are calculated to be 1e18.
- We have that `nominalLiq0 = (1 << 64) / 1e18 = 18` (same for `nominalLiq1`).
- `mintNominalLiq` adjust for rounding errors, giving `mintNominalLiq = 18 - 2 * 2 = 14`.
- The real liquidity available is 0, though, as the 1 wei cannot be split into the two ranges.

As a result, the `mint()` and `burn()` functions will revert. The users will be forced to donate to the contract an amount of token0 or token1 such that the calculation of the nominal liquidity is not overestimated. Note, however, that a malicious attacker can also front-run their transaction by donating an amount that will cause the transaction to revert, causing a denial of service attack.

Proof of concept

Modify the `forkSetup()` function in `test/single/Burve.t.sol` to use multiple ranges:

```diff
-       TickRange[] memory v3Ranges = new TickRange[](1);
-       v3Ranges[0] = TickRange(
-           clampedCurrentTick - v3RangeWidth,
-           clampedCurrentTick + v3RangeWidth
-       );
-
-       uint128[] memory v3Weights = new uint128[](1);
-       v3Weights[0] = 1;

+       TickRange[] memory v3Ranges = new TickRange[](2);
+       v3Ranges[0] = TickRange(
+           clampedCurrentTick - v3RangeWidth,
+           clampedCurrentTick + v3RangeWidth
+       );
+       v3Ranges[1] = TickRange(
+           clampedCurrentTick + v3RangeWidth,
+           clampedCurrentTick + v3RangeWidth + tickSpacing * 5
+       );
+
+       uint128[] memory v3Weights = new uint128[](2);
+       v3Weights[0] = 1;
+       v3Weights[1] = 1;
```

Add the following test function:

```solidity
function test_burnRevert() public forkOnly {
    deal(address(token0), address(alice), 1e32);
    deal(address(token1), address(alice), 1e32);

    // 1. Mint
    vm.startPrank(alice);
    token0.approve(address(burveV3), type(uint128).max);
    token1.approve(address(burveV3), type(uint128).max);
    burveV3.mint(address(alice), 2e18, 0, type(uint128).max);
    vm.stopPrank();

    // 2. Accumulate fees
    (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();
    deal(address(token0), address(this), 100_100_000e18);
    deal(address(token1), address(this), 100_100_000e18);

    pool.swap(
        address(this),
        true,
        100_100_000e18,
        TickMath.MIN_SQRT_RATIO + 1,
        new bytes(0)
    );
    pool.swap(
        address(this),
        false,
        100_100_000e18,
        sqrtPriceX96,
        new bytes(0)
    );
    vm.roll(block.timestamp * 100_000);

    // 3. Burn
    vm.prank(alice);
    vm.expectRevert(bytes("STF"));
    burveV3.burn(1e18, 0, type(uint128).max);
}
```

Run the test with the following command:

```
forge test --mt test_burnRevert --fork-url https://rpc.berachain.com/
// --fork-block-number 2109308 -vvvv
```

If we examine the logs, we can see that 24080586665097 token0 was collected. 16105268222585 token0 is transferred for the first range, and the transaction fails when trying to transfer 7975318442513 token0 for the second range. The problem is that the liquidity is overestimated, and a total of 24080586665098 token0 (1 over real balance) is required to mint the calculated liquidity.

Recommendations

A possible solution would be providing an off-chain service calculate the value of the liquidity off-line, so that the caller of `mint()` and `burn()` submits a liquidity amount, and the contract verifies that the provided liquidity is the optimal amount.

[H-02] First deposit front-running attack

Severity

Impact: High

Likelihood: Medium

Description

The `Burve` contract is subject to a common issue of ERC-4626 vaults, where the first deposit is front-run by an attacker.

1. A user will add liquidity into the contract to mint shares.
2. The attacker front-runs the user's transaction by minting one share and donating 50% of the liquidity provided by the user.
3. The user transaction is executed, and the shares minted are calculated as `mintNominalLiq * totalShares / totalNominalLiq`. As `compoundV3Ranges()` is executed, `totalNominalLiq` is inflated by the attacker's donation, so the division is truncated, and the user mints only 1 share: `x * 1 / (0.5x + 1) = 1`.

4. The attacker burns his share, receiving half of the liquidity of the contract.

Proof of concept

Add the following code to the file `test/single/Burve.t.sol` and run `forge test --mt test_firstDepositorDonation --fork-url https://rpc.berachain.com/ --fork-block-number 2073450`.

```
function test_firstDepositorDonation() public forkOnly {
    deal(address(token0), address(alice), 10e18);
    deal(address(token1), address(alice), 10e18);
    deal(address(token0), address(charlie), 10e18);
    deal(address(token1), address(charlie), 10e18);

    // Alice provides 1 wei of liquidity
    uint128 liq = 1;
    (int24 lower, int24 upper) = burveV3.ranges(0);
    (uint256 mint0, uint256 mint1) = getAmountsForLiquidity
      (liq, lower, upper, true);

    vm.startPrank(alice);
    token0.approve(address(burveV3), mint0);
    token1.approve(address(burveV3), mint1);
    burveV3.mint(address(alice), liq, 0, type(uint128).max);
    vm.stopPrank();

    // Alice donates 1e18 of liquidity
    liq = 1e18;
    (mint0, mint1) = getAmountsForLiquidity(liq, lower, upper, true);

    vm.startPrank(alice);
    token0.transfer(address(burveV3), mint0);
    token1.transfer(address(burveV3), mint1);
    vm.stopPrank();

    // Charlie provides 2e18 of liquidity and mints only 1 share
    liq = 2e18;
    (mint0, mint1) = getAmountsForLiquidity(liq, lower, upper, true);

    vm.startPrank(charlie);
    token0.approve(address(burveV3), mint0);
    token1.approve(address(burveV3), mint1);
    burveV3.mint(address(charlie), liq, 0, type(uint128).max);
    vm.stopPrank();

    assertEq(burveV3.totalShares(), 2);
    assertEq(burveV3.balanceOf(charlie), 1);
}
```

Recommendations

There are different ways to mitigate this issue, including:

- Use of dead shares: Forces to provide liquidity on the contract's deployment. This one would be the easiest to implement.
- Use of virtual deposit: OpenZeppelin's ERC4626 implements this solution and is well documented.

[H-03] Island shares ownership not updated on Burve token transfer

Severity

Impact: High

Likelihood: Medium

Description

When a Kodiak island contract is set in the `Burve`, on `mint()` a portion of the liquidity will be used to mint island shares, which will then be deposited into the station proxy at the name of the recipient, and the `islandSharesPerOwner` mapping will be updated for the recipient. The recipient will also receive `Burve` tokens (shares) for the liquidity provided to the Uniswap pools.

However, if the recipient transfers the `Burve` tokens to another address, the island shares are still assigned to him, both in `StationProxy` and in the `Burve`'s `islandSharesPerOwner` mapping. This means that after the transfer, the new owner of the `Burve` tokens will not be able to harvest the rewards in `StationProxy`, nor will he be able to burn the island shares in exchange for the underlying liquidity.

Recommendations

Overwrite the `_update()` function so that on `Burve` transfers:

- The proportional amount of LP tokens are withdrawn from `StationProxy`.
- The LP tokens are deposited again in the name of the new owner.
- `islandSharesPerOwner` is updated by both the old and new owner.

8.2. Medium Findings

[M-01] `EnsureClosure` would not add a vertex's closure if added in future

Severity

Impact: High

Likelihood: Low

Description

Assume there is a closureId which is supposed to include 3 vertexId , but for now only 2 of those are registered in the Token.sol . This is fine since when liquidity is added with `addLiq()` this is how it is done to ensure closure ->

```
for (uint8 i = 0; i < n; ++i) {
            VertexId v = newVertexId(i);
            if (cid.contains(v)) {
                Vertex storage vert = Store.vertex(v);
                // We need to add it to the Vertex so we can use it in swaps.
                vert.ensureClosure(cid);
```

Therefore , `homSet` and `homs` would be set for the first 2 vertices.

Now in the future lets say the third token was added as a new vertex and this token is included in our original closureId above , therefore when a user adds liquidity now the same piece of code would be invoked but the for the third vertex `homSet` and `homs` would not be set because in the `ensureClosure()` function ->

```
if (closure.contains(neighbor)) {
                if (self.homSet[neighbor][closure]) {
                    // We've already added this closure
                    return;
                }
```

Since for the first token/vertex the `homSet` has already been set , we would return instead of adding the third vertex's state.

Recommendations

Instead of returning to the `ensureClosure()` function, use `continue` instead.

[M-02] `FeeLib` calculations can revert

Severity

Impact: Low

Likelihood: High

Description

The `FeeLib` library is based on the Uniswap's V3 `PositionValue` library. However, it is not taken into account that the fee calculations expect overflows and underflows. This is not an issue in the original library, as it uses a solc version lower than 0.8.0, which does not revert to overflow/underflow. But in the Burve implementation, this will cause the transaction to revert.

The issue can be reproduced by running the `test_QueryValue_V3_NoFees` test on a fork of the Berachain network:

```
forge test --mt test_QueryValue_V3_NoFees --fork-url https://rpc.berachain.com/
// --fork-block-number 2073480
```

As a result, all the query functions in `Burve.sol` might revert unexpectedly.

Recommendations

Adapt the `FeeLib` library to use unchecked math operations, allowing the library to handle overflows and underflows without reverting the transaction.

[M-03] `setAdjustor` selector not included from `SimplexFacet`

Severity

Impact: Medium

Likelihood: Medium

Description

The `SimplexFacet` contract is missing the `setAdjustor` selector from the `simplexSelectors` array. This means that the `setAdjustor` function cannot be called.

Recommendations

```diff
-        bytes4[] memory simplexSelectors = new bytes4[](9);
+        bytes4[] memory simplexSelectors = new bytes4[](10);
         simplexSelectors[0] = SimplexFacet.getVertexId.selector;
         simplexSelectors[1] = SimplexFacet.addVertex.selector;
         simplexSelectors[2] = SimplexFacet.getTokens.selector;
         simplexSelectors[3] = SimplexFacet.getIndexes.selector;
         simplexSelectors[4] = SimplexFacet.numVertices.selector;
         simplexSelectors[5] = SimplexFacet.withdrawFees.selector;
         simplexSelectors[6] = SimplexFacet.setDefaultEdge.selector;
         simplexSelectors[7] = SimplexFacet.setName.selector;
         simplexSelectors[8] = SimplexFacet.getName.selector;
+        simplexSelectors[9] = SimplexFacet.setAdjustor.selector;
```

8.3. Low Findings

[L-01] No check for duplicated island range

The contractor of the `Burve` contract performs a series of checks over the setup data. These include a validation over the range values and a validating that the island range $(0, 0)$ is only used if the island address is provided.

However, it is not checked if the island range is used more than once. Being the island range the default value, it is recommended to check that it is only used once.

```
+   bool islandRangeUsed = false;
    for (uint256 i = 0; i < _ranges.length; ++i) {
        TickRange memory range = _ranges[i];

        ranges.push(range);

-       if (range.isIsland() && address(island) == address(0x0)) {
-           revert NoIsland();
+       if (range.isIsland()) {
+           if (islandRangeUsed) {
+               revert IslandRangeDuplicated();
+           } else if (address(island) != address(0x0)) {
+               revert IslandAddressProvided();
+           }
+           islandRangeUsed = true;
        }

        if (
            (range.lower % tickSpacing != 0) ||
            (range.upper % tickSpacing != 0)
        ) {
            revert InvalidRange(range.lower, range.upper);
        }
    }
```

[L-02] Importing interfaces from the `forge-std/interfaces` for production is not recommended

The `DecimalAdjustor` contract directly imports the `IERC20.sol` from the `forge-std/interfaces` as per the natspec comment given.

```
// Open zeppelin IERC20 doesn't have decimals for some reason.
import {IERC20} from "forge-std/interfaces/IERC20.sol";
```

`forge-std` is a collection of helpful utilities and contracts for use with Foundry, a smart contract development and testing toolchain. Using forge-std/interfaces directly in your production code is not recommended due to Dependency Versioning and Compatibility issues.

Recommended to use a `Stable, Widely-Used Library` such as the `IERC20Metadata.sol` interface from OpenZeppelin which implements the `decimals()` function.

[L-03] Missing sanity check for `defaultEdge` correctness

In cases where the edge between tokens is not set, the `edge()` function will return a default edge. However, this default edge is not properly checked for correctness. Specifically, when `defaultEdge.amplitude == 0`, the `edge()` function should `revert` to prevent using an invalid edge, as it is used in the `SwapFacet._preSwap()` and `LiqFacet.addLiq()` functions this could lead to unexpected behavior.

The issue arises from the removal of the proper check in the `Edge.getSlot0()` function, which was never reintroduced in the `Store::edge()` function. Additionally, the removal of this check leaves the `error NoEdgeSettings(address token0, address token1);` unused in the code.

Consider adding the missing sanity check to the `Store::edge()` function.

[L-04] Inclusion of test-only `ViewFacet` facet

A test-only `ViewFacet` facet is being included directly in the production code, which could potentially be used by external contracts as a source of truth. For example, the `ViewFacet.getVertex()` function returns only the first possible vertex for test-only usage.

The `ViewFacet` should not be included in production. It should only be added in tests to the SimplexDiamond `production` diamond cut by using the `diamondCut` function and adding it for testing purposes only.

Consider removing the `ViewFacet` facet from the SimplexDiamond initial diamond cut.

[L-05] `RemoveLiq` does not check if the vertex is locked

When removing liquidity with `removeLiq()` in the LiqFacet we fetch the vertexId if the vertex lies within the closure (has non zero balance) ->

```
for (uint8 i = 0; i < n; ++i) {
            VertexId v = newVertexId
            //(i);//AUDIT-no check if the vertex is locked.
            VaultProxy memory vProxy = VaultLib.getProxy(v);
            uint128 bal = vProxy.balance(cid, false);
```

But it is possible that the vertexId is currently locked, in which case withdrawals from the vertex's closure should be locked too.

Recommendations

Check if the vertex is locked when removing liquidity.

[L-06] Kodiac Island pause mechanism can lock funds

The Kodiac Island contract has a pause mechanism that disables the `mint()` and `burn()` functions, as well as a `restrictedMint` parameter that restricts minting capabilities to the manager.

When the `Burve` contract is using the island contract and any of the mentioned mechanisms are active, this will cause the `burn()` and/or `mint()` functions to revert. This is especially important in the case of the `burn()` function, as users might get their funds locked.

Recommendations

Allow skipping the mint and burn of island shares when the island contract is paused or restricted and provide a mechanism to complete the island shares minting and burning when the restrictions are lifted.