verichains

SECURITY AUDIT OF

BROWNFI AMM SMART CONTRACTS

BrownFi

Public Report

Oct 18, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Oct 18, 2024. We would like to thank the BrownFi for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the BrownFi AMM Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team identified several vulnerabilities in the source code and provided some recommendations. The BrownFi team has acknowledged and resolved most of these issues in the draft reports.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About BrownFi AMM Smart Contracts

BrownFi introduces a new spot AMM model based on oracle price, to offer high capital efficiency, flexible market making, while keeping simple UX, fungibility & reusability of LP tokens of Uniswap V2. The core concept of BrownFi AMM employs an elastic Parameterization of Limit Order-Book (PLOB) from a *published research papers* on IEEE Access, a notable scientific journal.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the BrownFi AMM Smart Contracts. It was conducted on the following GitHub repositories:

Repository	Commit
https://github.com/BrownFi/brownfi-periphery-evm	4a937bd530414b323d5570ea7e48208e3cf5cb02
https://github.com/BrownFi/brownfi-core-evm	8a403b5deb3cee3e05f3a1148c141f424a0810dc

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
e8bd4a19c72a9996cefb40fadb9e3cb30dfacc1ea056ef7f5cb807786e917709	./brownfi-periphery-evm-master/contracts/adapters/PythPriceFeed.sol
be8363a8ea695c03114aadf8c980b5c46d489dd985e514bc2c0e7064d79231ae	./brownfi-periphery-evm-master/contracts/interfaces/IPyth.sol
6c3d4020c16d59ddfcbc4d63637b2f3da7d89889e9c6e0ddf3ec53741a2d9059	./brownfi-periphery-evm-master/contracts/interfaces/PythStructs.sol
cb99cd9d40c51445a8908198a6a20404d97ce8ba8913662ef378c54cf90f6bbd	./brownfi-periphery-evm-master/contracts/interfaces/AggregatorV3Interface.sol
60760053849b916b72386fef1126ab69c7482c2aa6b5fb836368eff42905cb30	./brownfi-periphery-evm-master/contracts/interfaces/IWETH.sol

e59adc1f944a923765a1560c89a1193638ebbd21dfcffe83eea8e4d3cc4f9df8	./brownfi-periphery-evm-master/contracts/interfaces/IBrownFiV1Router03.sol
c3a00e91e4f5dff2cb2c7f8bba84a76c3a29ae1b3acec057933546a73d691102	./brownfi-periphery-evm-master/contracts/interfaces/IBrownFiV1Router02.sol
9c652bbf2e0cf0cc7967109113dec0c82c0c867570e5595c44e5abec361d2847	./brownfi-periphery-evm-master/contracts/interfaces/IPythPriceFeed.sol
2b63f199f838028184efefbcfd6cf2b9192624c3dae5dc1116ecbb15c36a67e8	./brownfi-periphery-evm-master/contracts/interfaces/IERC20.sol
42e7a55592abc103292fa37dfe7d8aa188866cb4623b261b606b2e6257df1d10	./brownfi-periphery-evm-master/contracts/interfaces/IBrownFiV1Router01.sol
6a93c99336a671d06b6d88d26a77c8abbef90dd470385c1b19962a58d07ef712	./brownfi-periphery-evm-master/contracts/BrownFiV1Router03.sol
0b1cec27d51f35e450cece4034e02de6098eff9ebc96dbde1bf8122e918d9861	./brownfi-periphery-evm-master/contracts/ultils/Multicall2.sol
2d858addbd734b0fbbf57caaa0956690ca1b69e2de5eb27f0f017ded5cfeabb6	./brownfi-periphery-evm-master/contracts/libraries/FullMath.sol
32cd65aadb2b5c447eb5dd60bee238b06eb847f5de49b7841a829c55f06b6814	./brownfi-periphery-evm-master/contracts/libraries/SafeMath.sol
de094a87d2c8c28da72f30386d8e11bb763d842b578751034ac8f3d5526e13bf	./brownfi-periphery-evm-master/contracts/libraries/BrownFiV1Library.sol
6712b3852c6df49832491ce5c456ed14d17f3299820ff261d0c7604742ac5e59	./brownfi-periphery-evm-master/contracts/BrownFiV1Router02.sol
bd2f4474e8b3648d65685142cd47d919c012b0daa56f5168f4174ec299236e33	./brownfi-core-evm-master/contracts/BrownFiV1Pair.sol
8c219ac6639db18552d64c3075591374d8df994b909d9c6c5834ed71d05f22ce	./brownfi-core-evm-master/contracts/interfaces/IBrownFiV1Factory.sol
cb99cd9d40c51445a8908198a6a20404d97ce8ba8913662ef378c54cf90f6bbd	./brownfi-core-evm-master/contracts/interfaces/AggregatorV3Interface.sol

6ba783539da56c73dc8b244101f3c607047c273509de13a643269e0adfb9c7d8	./brownfi-core-evm-master/contracts/interfaces/IBrownFiV1Callee.sol
fccf79ed9c813dd9e9bf673585db79192f95662fff7282792fce0369c6a48974	./brownfi-core-evm-master/contracts/interfaces/IBrownFiV1Pair.sol
2b63f199f838028184efefbcfd6cf2b9192624c3dae5dc1116ecbb15c36a67e8	./brownfi-core-evm-master/contracts/interfaces/IERC20.sol
8b2156f1b553f3ff6d17c5372569b8975711fef0e0328e29d808b9f2ab766a20	./brownfi-core-evm-master/contracts/interfaces/IBrownFiV1ERC20.sol
dac73aa47b3eb2f0900e96be982ef59dd2e90870cba9d048a79dc8a7b61fd4e9	./brownfi-core-evm-master/contracts/BrownFiV1Factory.sol
e4a9d451964a0689be2b244322a353de143ca4248d8736d91aca4ffadca4325f	./brownfi-core-evm-master/contracts/libraries/Math.sol
6633b57b0723b1d72e08cc3e8b29f0af838294e59863b6cdcce95a141ed02cdb	./brownfi-core-evm-master/contracts/libraries/UQ112x112.sol
4b1c95ff75de7342e0fadff58064820a4eb7c2fcb422a75b4994980ce8e216ae	./brownfi-core-evm-master/contracts/libraries/SafeMath.sol
cbc0c210294d2fa1ee1446f160e7503016d00f4fdf2e7b39d3e5964ec8abf222	./brownfi-core-evm-master/contracts/BrownFiV1ERC20.sol

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence

- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

BrownFi acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. BrownFi understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, BrownFi agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the BrownFi will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the BrownFi, the final report will be considered fully accepted by the BrownFi without the signature.

2. AUDIT RESULT

2.1. Overview

The BrownFi AMM Smart Contracts is based on the Uniswap V2 AMM, with significant modifications to the constant product invariant and the use of the Pyth oracle to maintain a swap price.

2.2. Findings

During the audit process, the audit team found some issues and recommendations in the given version of BrownFi AMM Smart Contracts.

#	Issue	Severity	Status
1	Integer underflow while calculating the delta value	CRITICAL	Fixed
2	Integer overflow in `fetchOraclePrice` function	CRITICAL	Fixed
3	Mismatch in swap formula between Pair and Router contracts	CRITICAL	Fixed
4	Admin can withdraw user tokens in the BrownFiV1Pair contract	HIGH	Fixed
5	The BrownFi pool cannot be rebalanced due to dependency on external price oracle	MEDIUM	Ack
6	Native ETHs for price update fee is not checked	LOW	Fixed
7	Accumulated price cannot be used	LOW	Fixed
8	Mint fee calculation based on the old formula of constant product	LOW	Fixed
9	Missing limitation for kappa parameter	LOW	Fixed
10	Deprecated functions based on the old invariant	INFORMATIVE	Fixed

2.2.1. Integer underflow while calculating the delta value CRITICAL

Affected files:

- brownfi-periphery-evm-master/contracts/libraries/BrownFiV1Library.sol

In the `delta` function, the expression `P * x - dy` may result in a negative value, which will cause an integer underflow in the `temp1` variable if calling `.sub(amountIn)`, which makes the `delta` function reverted.

```
function delta(uint amountIn, uint reserveOut, uint kappa, uint oPrice, bool isSell) public
pure returns (uint _delta) {
    uint temp1;
    uint temp2;
    if (isSell) {
        // temp1 = (P * dx - y)^2
        if (FullMath.mulDiv(oPrice, amountIn, Q128) < reserveOut) {
            temp1 = reserveOut.sub(FullMath.mulDiv(oPrice, amountIn,
Q128)).mul(reserveOut.sub(FullMath.mulDiv(oPrice, amountIn, Q128)));
        } else {
            temp1 = FullMath.mulDiv(oPrice, amountIn,
Q128).sub(reserveOut).mul(FullMath.mulDiv(oPrice, amountIn, Q128).sub(reserveOut));
        }
    } else {
        // temp1 = (P * x - dy)^2
        // AUDIT: INTEGER OVERFLOW
        temp1 = FullMath.mulDiv(oPrice, reserveOut,
Q128).sub(amountIn).mul(FullMath.mulDiv(oPrice, reserveOut, Q128).sub(amountIn));
    }
    // temp2 = 2 * P * K * y * dx
    temp2 = FullMath.mulDiv(oPrice, amountIn, Q128).mul(FullMath.mulDiv(kappa, reserveOut,
Q128)).mul(2);
    _delta = temp1.add(temp2);
    return _delta;
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team.

2.2.2. Integer overflow in `fetchOraclePrice` function CRITICAL

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol
- brownfi-periphery-evm-master/contracts/libraries/BrownFiV1Library.sol

In the `fetchOraclePrice` function in both the `BrownFiV1Pair` and `BrownFiV1Library` contracts, when the value of `decimalShift` is negative, the expression `uint(decimalShift)` will cause an integer overflow. This can result in an incorrect oracle price, which may lead to further exploitation. Also, the duplicated code in the `fetchOraclePrice` function should be refactored to avoid redundancy.

```
// Feed oracle price from chainlink
function fetchOraclePrice() internal view returns (uint) {
    // ...
    // get decimals from oracle
    uint8 decimals = _priceFeed.decimals();
    // convert price to Q128 base on decimals
    if (qti == 0) {
        if (decimalShift > 0) {
            return FullMath.mulDiv(10**uint(decimals) * 10**uint(decimalShift), Q128,
uint(price));
        } else if (decimalShift < 0) {
            return FullMath.mulDiv(10**uint(decimals), Q128, uint(price) *
10**uint(decimalShift)); // AUDIT: INTEGER OVERFLOW
        } else {
            return FullMath.mulDiv(10**uint(decimals), Q128, uint(price));
        }
    }
    else {
        if (decimalShift > 0) {
            return FullMath.mulDiv(uint(price), Q128, 10**uint(decimals) *
10**uint(decimalShift));
        } else if (decimalShift < 0) {
            return FullMath.mulDiv(uint(price) * 10**uint(decimalShift), Q128,
10**uint(decimals)); // AUDIT: INTEGER OVERFLOW
        } else {
            return FullMath.mulDiv(uint(price), Q128, 10**uint(decimals));
        }
    }
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team.

2.2.3. Mismatch in swap formula between Pair and Router contracts CRITICAL

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol
- brownfi-periphery-evm-master/contracts/libraries/BrownFiV1Library.sol

The `getAmountOut` function shows that the formula for calculating the output amount of the other asset differs from the one used in the `checkInventory` function of the `BrownFiV1Pair` contract. This mismatch could lead to unexpected outcomes during token swaps. If the amount estimated by `getAmountOut` is less than the actual amount calculated by `checkInventory`, users may receive less than expected. Conversely, if `getAmountOut` estimates a greater amount than `checkInventory` calculates, the swap transaction will revert.

```
// given an input amount of an asset and pair reserves, returns the maximum output amount
of the other asset
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut, uint kappa, uint
oPrice, uint fee, bool zeroForOne) internal pure returns (uint amountOut) {
    require(amountIn > 0, 'BrownFiV1Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'BrownFiV1Library: INSUFFICIENT_LIQUIDITY');
    if (kappa == Q128.mul(2)) {
        if (zeroForOne) {
            // dy = P * y * dx / (P * dx + y)
            amountOut = FullMath.mulDiv(oPrice, reserveOut, Q128).mul(amountIn) /
FullMath.mulDivRoundingUp(oPrice, amountIn, Q128).add(reserveOut);
        } else {
            // dx = (x * dy) / (P * x + dy)
            amountOut = amountIn.mul(reserveOut) / FullMath.mulDivRoundingUp(oPrice,
reserveOut, Q128).add(amountIn);
        }
        amountOut = FullMath.mulDiv(amountOut, FEE_DENOMINATOR - fee, FEE_DENOMINATOR);
    } else {
        // ...
    }
}
```

One thing to note is that `kappa` is not used in the `BrownFiV1Pair` contract, which suggests the contract may be incomplete or not fully implemented. Please provide the complete contract if it is available.

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team by switching to the new formula.

2.2.4. Admin can withdraw user tokens in the BrownFiV1Pair contract HIGH

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol

The `isVerify` flag is used to determine whether to check the post-trade inventory against the pre-trade inventory. If `isVerify` is set to `true`, the contract will revert if the post-trade inventory is less than the pre-trade inventory. However, the `setIsVerify` function allows the admin to disable this check, which means the admin can withdraw user tokens from the contract without any restrictions.

```
function checkInvetory(
    uint amount0Out,
    uint amount1Out,
    uint balance0,
    uint balance1,
    uint112 _reserve0,
    uint112 _reserve1
) private returns (uint) {
    uint oPrice = fetchOraclePrice();
    Inventories memory i;
    // calculate pre-trade invetory x*P + y
    i.preInvetory = FullMath.mulDiv(oPrice, _reserve0, Q128).add(_reserve1);
    _update(balance0, balance1, _reserve0, _reserve1);
    // calculate post-trade invetory (x - dx)*P + (y + dy)
    if (amount0Out > 0) {
        uint amount0OutWithoutFee = FullMath.mulDiv(amount0Out, FEE_DENOMINATOR,
FEE_DENOMINATOR - fee);
        i.postInvetory = FullMath.mulDiv(oPrice, _reserve0 - amount0OutWithoutFee,
Q128).add(reserve1);
    } else {
        uint amount1OutWithoutFee = FullMath.mulDiv(amount1Out, FEE_DENOMINATOR,
FEE_DENOMINATOR - fee);
        i.postInvetory = FullMath.mulDiv(oPrice, reserve0, Q128).add(_reserve1 -
amount1OutWithoutFee);
    }
    if (isVerify) { // AUDIT: admin can withdraw user tokens from the contract
        require(i.postInvetory >= i.preInvetory, 'BrownFiV1: INVALID_TRADE');
    }
    return oPrice;
}

function setIsVerify(bool _isVerify) external {
    require(msg.sender == IBrownFiV1Factory(factory).feeToSetter(), 'BrownFiV1:
FORBIDDEN');
    isVerify = _isVerify;
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team.

2.2.5. The BrownFi pool cannot be rebalanced due to dependency on external price oracle MEDIUM

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol

Looking at the source code of the BrownFiV1Pair contract, we can see that the original AMM constraint from Uniswap V2 has been replaced with the following new formula:

```
function swap(
    uint amount0Out,
    uint amount1Out,
    address to,
    bytes calldata data
) external lock {
    // ...
    uint amount0In = b.balance0 > _reserve0 - amount0Out ? b.balance0 - (_reserve0 -
amount0Out) : 0;
    uint amount1In = b.balance1 > _reserve1 - amount1Out ? b.balance1 - (_reserve1 -
amount1Out) : 0;
    require(amount0In > 0 || amount1In > 0, 'BrownFiV1: INSUFFICIENT_INPUT_AMOUNT');

    // verify post-trade invetory is greater or equal to pre-trade invetory
    uint swapPrice = checkInvetory(amount0Out, amount1Out, b.balance0, b.balance1,
_reserve0, _reserve1);
    emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to, swapPrice);
}

function checkInvetory(
    uint amount0Out,
    uint amount1Out,
    uint balance0,
    uint balance1,
    uint112 _reserve0,
    uint112 _reserve1
) private returns (uint) {
    uint oPrice = fetchOraclePrice();
    Inventories memory i;
    // calculate pre-trade invetory x*P + y
    i.preInvetory = FullMath.mulDiv(oPrice, _reserve0, Q128).add(_reserve1);
    _update(balance0, balance1, _reserve0, _reserve1);
    // calculate post-trade invetory (x - dx)*P + (y + dy)
    if (amount0Out > 0) {
        uint amount0OutWithoutFee = FullMath.mulDiv(amount0Out, FEE_DENOMINATOR,
FEE_DENOMINATOR - fee);
        i.postInvetory = FullMath.mulDiv(oPrice, _reserve0 - amount0OutWithoutFee,
Q128).add(reserve1);
    } else {
        uint amount1OutWithoutFee = FullMath.mulDiv(amount1Out, FEE_DENOMINATOR,
FEE_DENOMINATOR - fee);
        i.postInvetory = FullMath.mulDiv(oPrice, reserve0, Q128).add(_reserve1 -
amount1OutWithoutFee);
    }
    if (isVerify) {
        require(i.postInvetory >= i.preInvetory, 'BrownFiV1: INVALID_TRADE');
    }
    return oPrice;
}
```

In case the user inputs token Y and receives token X, the new formula can be expressed as follows:

$$
\begin{aligned}
&x : reserve\ of\ tokenX\\
&y : reserve\ of\ tokenY\\
&P : oracle\ price\ of\ x/y
\end{aligned}
$$

$$
\begin{aligned}
&(x-dx)P+(y+dy)\geq xP+y\\
\Leftrightarrow\quad &-dxP+dy\geq 0\\
\Leftrightarrow\quad &dx\leq\frac{dy}{P}
\end{aligned}
$$

The formula means that, regardless of the reserves of token X and token Y, the output amount of token X is calculated based on the oracle price P and the input amount of token Y.

The issue is that the output price is not determined by the reserve ratio of the two tokens in the pool, so there is no asset rebalancing mechanism. If the reserve of one token becomes too low, the pool will be considered out of liquidity. However, adding liquidity relies on the reserve ratio of the two tokens (this is the original design of Uniswap V2), which is ineffective as it doesn't help rebalance the pool.

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged by the BrownFi team but has not been fixed yet. As stated by the BrownFi team: *Serving as a shared concentrated liquidity pool, BrownFi AMM can become "imbalanced" when one side of the liquidity is significantly smaller than the other. To prevent no-fee swap exploitation by adding/removing single-sided liquidity, BrownFi currently has no built-in rebalancing mechanism. In fact, pool imbalance is common across all AMMs with liquidity concentration (Uniswap V3, Maverick, TraderJoe, Curve, etc.). When the pool is imbalanced, similar to an Uniswap V3 position, LPers can either (1) wait for the market to return to the price range or (2) withdraw the LP and create another pool.*

2.2.6. Native ETHs for price update fee is not checked LOW

Affected files:

- brownfi-periphery-evm-master/contracts/BrownFiV1Router03.sol

In all swap functions in the `BrownFiV1Router03` contract, such as `swapExactETHForTokensWithPrice` and `swapExactTokensForETHWithPrice`, users are required to send native ETH to the contract to cover the price update fees. However, the contract does not check whether `msg.value` is greater than or equal to the update fee. This may result in situations where users use fees left inside the contract by previous users or send more ETH than required.

```
function swapExactTokensForTokensWithPrice(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline,
    bytes[] calldata priceUpdate
) external virtual override payable ensure(deadline) returns (uint[] memory amounts) {
    // update price
    IPyth(PYTH).updatePriceFeeds{value:
IPyth(PYTH).getUpdateFee(priceUpdate)}(priceUpdate);
    address pair = BrownFiV1Library.pairFor(factory, path[0], path[1]);
    amounts = BrownFiV1Library.getAmountsOut(factory, amountIn, path);
    require(amounts[amounts.length - 1] >= amountOutMin, 'BrownFiV1Router:
INSUFFICIENT_OUTPUT_AMOUNT');
    TransferHelper.safeTransferFrom(
        path[0], msg.sender, pair, amounts[0]
    );
    _swap(amounts, path, to);
}
```

RECOMMENDATION

We recommend adding a check to ensure that the `msg.value` is greater than or equal to the update fee before proceeding with the swap functions. Additionally, there should be a mechanism to recover any stuck ETH in the contract after a certain period of time.

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team.

2.2.7. Accumulated price cannot be used LOW

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol

Due to changes in the AMM formula, the accumulated price formula from Uniswap V2 in the `BrownFiV1Pair` contract can no longer be used to calculate the price of the two tokens. Specifically, the `price0CumulativeLast` and `price1CumulativeLast` variables should be removed from the contract to prevent potential usages that may lead to unexpected results.

```
function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1)
private {
    require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'BrownFiV1: OVERFLOW');
    uint32 blockTimestamp = uint32(block.timestamp % 2**32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) *
timeElapsed;
```

```
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) *
timeElapsed;
    }
    reserve0 = uint112(balance0);
    reserve1 = uint112(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team by removing the accumulated price calculation.

2.2.8. Mint fee calculation based on the old formula of constant product LOW

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol

The `_mintFee` function in the `BrownFiV1Pair` contract calculates the mint fee based on the old constant product formula. This formula is no longer suitable for the new AMM formula used in the contract and therefore cannot be used anymore.

```
function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
    address feeTo = IBrownFiV1Factory(factory).feeTo();
    feeOn = feeTo != address(0);
    uint _kLast = kLast; // gas savings
    if (feeOn) {
        if (_kLast != 0) {
            uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
            uint rootKLast = Math.sqrt(_kLast);
            if (rootK > rootKLast) {
                uint numerator = totalSupply.mul(rootK.sub(rootKLast));
                uint denominator = rootK.mul(5).add(rootKLast);
                uint liquidity = numerator / denominator;
                if (liquidity > 0) _mint(feeTo, liquidity);
            }
        }
    } else if (_kLast != 0) {
        kLast = 0;
    }
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team by removing the mint fee calculation.

2.2.9. Missing limitation for kappa parameter LOW

Affected files:

- brownfi-core-evm-master/contracts/BrownFiV1Pair.sol

The value of the kappa parameter in the setKappa function is not restricted as outlined in the documentation, which may lead to unexpected results if an invalid value is set.

```
function setKappa(uint _kappa) external {
    require(msg.sender == IBrownFiV1Factory(factory).feeToSetter(), 'BrownFiV1:
FORBIDDEN');
    kappa = _kappa; // AUDIT: missing limitation [0.001, 2]
}
```

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team by adding limitations for the kappa parameter.

2.2.10. Deprecated functions based on the old invariant INFORMATIVE

Affected files:

- brownfi-periphery-evm-master/contracts/libraries/BrownFiV1LiquidityMathLibrary.sol
- brownfi-periphery-evm-master/contracts/BrownFiV1Router02.sol

Most of the functions in the BrownFiV1LiquidityMathLibrary are copied from Uniswap V2, using the old constant product invariant formula. We suggest that the dev team review these functions and remove any unused ones to prevent incorrect usage in the future.

Some of these functions include: computeProfitMaximizingTrade, getReservesAfterArbitrage, and getLiquidityValueAfterArbitrageToPrice.

Even the addLiquidity and removeLiquidity functions in the BrownFiV1Router02 contract still use the old constant product invariant formula, which prevents liquidity providers from adding or removing liquidity to rebalance the pool. The pool cannot be rebalanced simply by removing all the liquidity and adding it back, as the current pool ratio is imbalanced. Additionally, it is not possible to create a new pool with the same token pair as the existing one.

RECOMMENDATION

It's better to review the whole codebase and remove the deprecated functions to avoid confusion and potential misuse.

UPDATES

- **Oct 18, 2024**: This issue has been acknowledged and fixed by the BrownFi team by removing two files `BrownFiV1LiquidityMathLibrary.sol` and `BrownFiV1OracleLibrary.sol`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Oct 18, 2024*	Public Report	Verichains Lab

Table 2. Report versions history