Security review

Infinix

From:
Stela Labs
Info@stelalabs.com

To:
Infinix finance

Infinix review

The following report is a result of a security review from **Stela Labs** (Info@stelalabs.com), requested by **Infinix** team to conduct a security check of their smart contracts. Fifteen days were spent on the review.

The review was made on a codebase with a following commit hash:
`96c4081af5dce8c668b41133e276bc561250b2d2` .

Summary

The codebase represents a fork of Prepetual protocol with a few changes:

- API3 oracle is used instead of Chainlink.
- Using TraderJoe (copy of Uniswap V2) exchange instead of Balancer.
- Using a logarithmic price of the underlying asset

The review was done assuming that the original codebase is secure; auditing it from scratch would take much more time and resources. The main efforts were focused on reviewing the changes and their impact on the system.

Issues

1. **HIGH**. **The token swaps are used without passing a proper `_minOutputTokenBought` value**.

   > ***Update:*** *The issue is addressed by creating a UniswapV2 oracle contract for every relevant token pair. Before executing the trade, it is verified that the price did not shift over a certain threshold. The oracle must be updated with the defined frequency. Otherwise, it will show the outdated average price. Additionally, the maximum slippage is constant and equals 10%.*

   When the insurance fund wants to make a trade on an exchange, it calls the `ExchangeWrapper` contract's `swapInput` function:

   ```
       function swapInput(
       IERC20 _inputToken,
       IERC20 _outputToken,
       Decimal.decimal calldata _inputTokenSold,
       Decimal.decimal calldata _minOutputTokenBought,
       Decimal.decimal calldata _maxPrice
   )
   ```

   As usual, this function has the `_minOutputTokenBought` and the `_maxPrice` parameters to prevent front-running and price manipulations. But when it is called, both parameters are passed with `Decimal.zero()` values.

   Because of that, all the trades can be manipulated by an MEV attack.

2. **MEDIUM**. **VWAP may not correspond to TWAP correctly**.

   > ***Update:*** *Not addressed. The team accepted the risk.*

   A difference between the futures price and the underlying price defines the funding payment. To prevent manipulation, the average price is used. Ideally, you would want to use the same algorithm

to calculate both average prices. When calculating the underlying price, the following time-weighted average algorithm is used:

```
 // premium = twapMarketPrice - vwapIndexPrice
    // timeFraction = fundingPeriod(1 hour) / 1 day
    // premiumFraction = premium * timeFraction
    Decimal.decimal memory underlyingPrice = getUnderlyingPrice();
    SignedDecimal.signedDecimal memory premium =

MixedDecimal.fromDecimal(getTwapPrice(spotPriceTwapInterval)).subD(underlyingPri

    SignedDecimal.signedDecimal memory premiumFraction =
premium.mulScalar(fundingPeriod).divScalar(int256(1 days));
```

API3 oracles are supposed to give the volume-weighted average price. The problem is that if these algorithms differ significantly, the average prices may also be different even if the actual price of futures and underlying asset are always the same. It is hard to evaluate the potential impact of this difference because it depends on many factors. But this issue can have a dangerous effect under some parameters and price movements.

3. **MEDIUM**. **An oracle price can be outdated.**

> ***Update:*** *The issue is fixed by checking that the price was updated at least the* `stalePriceThreshold` *ago.*

When getting a price from an oracle, it can also return a timestamp of the last update:

```
function readDataFeedWithDapiName(bytes32 dapiName)
    external
    view
    override
    returns (int224 value, uint32 timestamp)
{
    bytes32 dapiNameHash = keccak256(abi.encodePacked(dapiName));
    require(
        readerCanReadDataFeed(dapiNameHash, msg.sender),
        "Sender cannot read"
    );
    bytes32 dataFeedId = dapiNameHashToDataFeedId[dapiNameHash];
    require(dataFeedId != bytes32(0), "dAPI name not set");
    DataFeed storage dataFeed = dataFeeds[dataFeedId];
    return (dataFeed.value, dataFeed.timestamp);
}
```

But when this oracle is called, only the price is read and used. The timestamp is only checked to be non-zero.

```
function getPrice(bytes32 _dapiName) external view override returns (uint256)
{
    int224 value =
IDapiServer(dapiServer).readDataFeedValueWithDapiName(_dapiName);
```

```
    uint256 scaledVal;
    unchecked {
        scaledVal = uint256(int256(value)) * SCALE;
    }
    return scaledVal.log2();
}
```

There is a chance that the price is outdated and should be verified.

4. **MINOR. The `traderJoeSwapOut` function is giving a larger allowance to the router than it should.**

> ***Update:*** *The issue is fixed by giving less allowance (`expectedTokenInAmount`).*

Before executing a trade, the specific amount (`expectedTokenInAmount`) of tokens are transferred to the `ExchangeWrapper`:

```
Decimal.decimal memory expectedTokenInAmount = calcTraderJoeInGivenOut(
        address(_inputToken),
        address(_outputToken),
        _outputTokenBought
    );
    require(
        _maxInputTokenSold.cmp(expectedTokenInAmount) >= 0,
        "max input amount less than expected"
    );

    //___2 transfer input tokens to exchangeWrapper
    _transferFrom(_inputToken, sender, address(this), expectedTokenInAmount);
```

But the allowance is given for a different amount of tokens which is greater than the amount transferred to the contract:

```
_approve(IERC20(_inputToken), address(joeRouter), _maxInputTokenSold);
```

It may not have any direct attack vector, but if there is any other bug in the code, this extra allowance can help the attacker.

5. **TRUST. The oracle is trusted.**

As opposed to Chainlink, API3 is the first-party oracle. So the system relies on the data provider to update the price frequently and correctly.