Security Audit Report for Leverage Farming

Date: September 22, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.09.22 18:10:35 +08'00'

Contents

Report Manifest

Item	Description
Client	Extra Finance
Target	Leverage Farming

Version History

Version	Date	Description
1.0	September 22, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repo of the Leverage Farming smart contracts [1] of Extra Finance. Leverage Farming is a leveraged yield farming protocol that enables users to deposit funds, borrow extra funds through the lending pool, and invest in `Velodrome` liquidity pools [2].

During this audit, we operate under the following presumptions:

- The lending pool adopted by Leverage Farming is the contract [3] on the Optimistic Ethereum network.
- All `Velodrome`-related addresses used by Leverage Farming are official smart contracts.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Leverage Farming	`Version 1`	`b4e76554e3a14f00f99bbdda4ecaab6fdcda6eff`
	`Version 2`	`c339a997ba36b46614d8288364360510169e04d9`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the

[1] https://github.com/ExtraFi/contracts

[2] https://velodrome.finance/

[3] https://optimistic.etherscan.io/address/0xbb505c54d71e9e599cb8435b4f0ceec05fc71cbd

computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver

* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [4] and Common Weakness Enumeration [5]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[4] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[5] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **two** potential issues. Besides, we also have **two** recommendations and **four** notes.

- High Risk: 1
- Low Risk: 1
- Recommendation: 2
- Note: 4

ID	Severity	Description	Category	Status
1	High	Unexpected price impact during liquidation	DeFi Security	Fixed
2	Low	Unfair reward distribution	DeFi Security	Confirmed
3	-	Revise improper comments	Recommendation	Fixed
4	-	Remove unused code	Recommendation	Fixed
5	-	Potential centralization risks	Note	-
6	-	The protocol should not support *weird* ERC20 tokens	Note	-
7	-	Potential price manipulation risks	Note	-
8	-	Potential inaccurate calculation	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Unexpected price impact during liquidation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In this protocol, users have the ability to liquidate a position by repaying all debts and acquiring the liquidated position's LP tokens. The vault gives liquidators the right to claim all returned tokens if the total repaid value exceeds that of the removed liquidity.

```
if (state.repaidValue >= state.removedLiquidityValue) {
    // The debt repaid by the liquidator exceeds the value of the position,
    // when there is negative equity position due to delayed liquidation.
    // Then transfer all the left tokens to liquidate receiver
    state.liquidateFeeValue = 0;
    state.liquidatorReceive0 += state.amount0Left;
    state.liquidatorReceive1 += state.amount1Left;
    state.amount0Left = 0;
    state.amount1Left = 0;
} ...
```

Listing 2.1: VeloVaultPositionLogic.sol

However, both the repaid and removed liquidity values are calculated in terms of `token0` via the `valueOfTokensInToken0` function. If a position's debts consist entirely of `token0`, the price does not contribute into the repaid value calculation. On the other hand, estimating the value of the removed liquidity necessitates the use of price to convert `token1` to `token0`.

This implies that a manipulated price could differentially affect these two values. Considering the default 20% max price deviation, there exists a substantial potential to exploit this inconsistency for profit. For example, an attacker could manipulate the price to disadvantage the removed liquidity value calculation, making it smaller compared to the repaid value. This could unfairly allow the liquidation of the position to yield more assets than are rightfully due.

```
state.repaidValue = VeloPositionValue.valueOfTokensInToken0(
    state.amount0Repaid,
    state.amount1Repaid,
    state.price
);
state.liquidatorReceive0 = params.maxRepay0.sub(state.amount0Repaid);
state.liquidatorReceive1 = params.maxRepay1.sub(state.amount1Repaid);

// remove liqudity from amm
(state.amount0Left, state.amount1Left) = closePositionPartially(
    ClosePositionPartiallyParams(
        params.vaultPositionId,
        params.percent,
        params.minAmount0WhenRemoveLiquidity,
        params.minAmount1WhenRemoveLiquidity,
        params.deadline
    )
);
state.removedLiquidityValue = VeloPositionValue.valueOfTokensInToken0(
    state.amount0Left,
    state.amount1Left,
    state.price
);
```

Listing 2.2: VeloVaultPositionLogic.sol

Impact A malicious liquidator could get more assets from liquidation with the manipulated price.

Suggestion Reduce the maximum allowed price deviation.

2.1.2 Unfair reward distribution

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the protocol design, a vault is built on the top of a `Velodrome` pair and corresponding gauge. The vault deposits users' tokens into the pair and stakes the LP tokens in the gauge to earn rewards. However, this could potentially lead to an issue of unfair reward distribution due to `Velodrome`'s gauge reward mechanism.

`Velodrome` requires users to stake LP tokens for a specific time period to participate in reward distribution. However, the vault does not check the deposit time when calculating rewards. Hence, there is no way to confirm whether a user has staked for a sufficient duration to be eligible for gauge rewards.

Specifically, a malicious user could take the following steps to launch an attack:

- Borrow a flashloan and deposit it into the vault to open a position.
- Invoke the `distribute` function in `Velodrome`'s `Voter` contract to distribute rewards to the gauge.
- Close the position, which invokes the vault's `claimRewardsAndReInvestToLiquidityInternal` function. This claims rewards from the gauge and reinvests them. The reinvested liquidity is added to the vault's `totalLp`, inflating rewards for the malicious user. The inflated rewards are distributed to the malicious user immediately in this step.
- Repay the flashloan.

```
function removeLiquidityShares(
    VaultTypes.VeloVaultState storage vaultState,
    VaultTypes.VeloPosition storage position,
    uint256 lpShares
) internal returns (uint256 liquidity) {
    liquidity = lpShares.mul(vaultState.totalLp).div(
        vaultState.totalLpShares
    );

    vaultState.totalLp = vaultState.totalLp.sub(liquidity);
    vaultState.totalLpShares = vaultState.totalLpShares.sub(lpShares);

    position.lpShares = position.lpShares.sub(lpShares);
}
```

Listing 2.3: VeloVaultPositionLogic.sol

```
function claimRewardsAndReInvestToLiquidityInternal(
    uint256 vaultId
)
    internal
    returns (uint256 liquidity, uint256 fee0, uint256 fee1, uint256 rewards)
{
    VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
        .getVaultStorage();

    ClaimRewardsAndReInvestToLiquidityState memory state;

    rewards = claimRewards();

    (state.amount0, state.amount1) = swapRewardsToBaseToken(rewards);

    if (state.amount0 > 0 || state.amount1 > 0) {
        ...
        (state.amount0, state.amount1, liquidity) = VeloLiquidityLogic
            .swapAndAddLiquidity(
                VeloLiquidityLogic.AddLiquidityParam(
                    state.amount0 - fee0,
                    state.amount1 - fee1,
                    0,
                    0,
                    block.timestamp + 1
                )
```

```
            );
        }

        if (liquidity > 0) {
            ...
            IGaugeV2(vaultStorage.state.gauge).deposit(liquidity);
            vaultStorage.state.totalLp = vaultStorage.state.totalLp.add(
                liquidity
            );
        ...
        }
    }
```

Listing 2.4: VeloVaultRewardsLogic.sol

It is worth noting that in `Version 2`, the vault integrates a timelock mechanism that enforces a minimum time interval between a user's deposit and withdrawal. A user is required to wait at least 5 minutes (`MINIMAL_WITHDRAW_WAIT_TIME`) post-deposit to withdraw from the position. While this measure mitigates flashloan attacks, it does not completely eliminate the issue of unfair reward distribution.

```
    require(block.timestamp >= position.lastInvestTime + Constants.MINIMAL_WITHDRAW_WAIT_TIME, "5-
        Minute Lock After Adding To Position!");
```

Listing 2.5: VeloVaultPositionLogic.sol

Impact Malicious users can falsely claim rewards from gauges without properly staking for the required duration.

Suggestion Revise the withdrawal logic accordingly.

2.2 Additional Recommendation

2.2.1 Revise improper comments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several typos in the comments of some smart contracts, as follows:

- In the `VeloVaultV2` contract, there is a typo in the comment for the `adminSetVault` function: ”priviledge" should be ”privilege".

```
    /// notice Set vault with admin previledge
```

Listing 2.6: VeloVaultV2.sol

- In the `VaultFactory` contract, the comment for the `newVault` function mentions the `UniswapV3` pool but it should refer to the `Velodrome` pool instead.

```
    /// notice New a Vault which contains the uniswapV3 pool's info and the debt positions
```

Listing 2.7: VaultFactory.sol

Impact N/A

Suggestion Revise the comments accordingly.

2.2.2 Remove unused code

Status Fixed in Version 2

Introduced by Version 1

Description Unused code in some smart contracts can be removed, as follows:

- The RESOLUTION constant in the Precision library is unused and can be removed.

```
uint8 internal constant RESOLUTION = 96;
```

Listing 2.8: Precision.sol

- The quoteEarnedRewards function in the VeloVaultRewardsLogic library is declared but never used.

```
function quoteEarnedRewards() internal view returns (uint256 earned) {
    VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
        .getVaultStorage();

    earned = IGaugeV2(vaultStorage.state.gauge).earned(address(this));
}
```

Listing 2.9: VeloVaultRewardsLogic.sol

Impact N/A

Suggestion Remove unused code.

2.3 Note

2.3.1 Potential centralization risks

Description The Leverage Farming protocol exhibits a high degree of reliance on delegatecalls to library contracts, with the library addresses fetched from the AddressRegistry contract. This creates a single point of failure. If an attacker were to compromise the owner of the AddressRegistry contract, they could potentially incapacitate the entire system. Similarly, the VeloSwapPathManager contract is responsible for determining the swap path from reward tokens to base tokens. If an attacker gains control over this contract, they could manipulate the swap path. This scenario presents another centralization risk for the protocol.

Moreover, the owner of the VeloPositionManager contract possesses the capacity to alter critical configurations across all vaults. This, too, presents a centralization risk in the event that the owner's account is compromised.

Feedback from the Developers The owner of the above contracts is already transferred to a multisig account.

2.3.2 The protocol should not support *weird* ERC20 tokens

Description The lending pool adopted by the Leverage Farming protocol should only support underlying tokens that follow standard ERC20 specifications. Non-standard *weird* ERC20 tokens [1], such as deflation,

[1] https://github.com/d-xo/weird-erc20

inflation, rebasing, and callback-supporting tokens, may introduce potential security risks to the protocol. To mitigate these risks, the protocol should refrain from supporting such tokens.

Feedback from the Developers The lending pool does not support deflation/inflation/rebase tokens. As for other weird tokens like callback-support tokens, we need some time to confirm if the tokens currently listed have this type.

2.3.3 Potential price manipulation risks

Description The `swapRewardsToBaseToken` function in the `VeloVaultRewardsLogic` library swaps claimed reward tokens for either `token0` or `token1` on `Velodrome`, prioritizing a route to `token0`. If no route to `token0` is found, it swaps to `token1`.

However, the `amountOutMin` parameter (line 245) is not specified for the swap call, creating a price manipulation attack vector due to the absence of slippage control. This could allow an attacker to sandwich the unprotected swap for profit, resulting in a loss of rewards.

To mitigate this risk, the protocol aims to reinvest frequently to prevent excessive reward accumulation, thereby limiting the amount of rewards that can be manipulated and reducing potential risk.

```
187    function swapRewardsToBaseToken(
188        uint256 claimedRewards
189    ) internal returns (uint256 amount0Increased, uint256 amount1Increased) {
190        VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
191            .getVaultStorage();
192
193        if (claimedRewards >= MINIMAL_REWARDS_REINVEST) {
194            ...
204            IRouter.route[] memory swapRoute;
205            swapRoute = IVeloSwapPathManager(vaultStorage.swapPathManager)
206                .getPath(
207                    vaultStorage.rewardTokens[0],
208                    vaultStorage.state.token0
209                );
210            if (swapRoute.length == 0) {
211                swapRoute = IVeloSwapPathManager(vaultStorage.swapPathManager)
212                    .getPath(
213                        vaultStorage.rewardTokens[0],
214                        vaultStorage.state.token1
215                    );
216            }
217
218            require(swapRoute.length > 0, "no swap route for rewards!");
219            ...
242            uint[] memory amounts = IRouterV2(vaultStorage.veloRouter)
243                .swapExactTokensForTokens(
244                    claimedRewards,
245                    0,
246                    routes,
247                    address(this),
248                    block.timestamp + 1
249                );
250            ...
```

```
        }
    }
```

Listing 2.10: VeloVaultRewardsLogic.sol

Feedback from the Developers This issue is not very severe because the reinvestment frequency is high and the rewards required to swap are relatively small, so there is not enough room for arbitrage. Since the current contract parameters cannot be modified, it is not possible to add new slippage parameters for rewards swaps. Therefore, we will maintain the original logic. To mitigate the risk of sandwich arbitrage, we will use a high reinvestment frequency to ensure that the reward amount need to swap is minimal, thus reducing the risk.

2.3.4 Potential inaccurate calculation

Description The `getSwapAmountForAddLiquidity` function in the `VeloLiquidityMath` library calculates the fair amount of tokens required to add liquidity to a `Velodrome` pair by simulating changes to the pair's reserves. However, it updates the amounts using the `getAmountOut` function in the `Velodrome` pair contract, which references the reserves prior to the simulation. This discrepancy between the simulated and actual reserves can result in inaccurate calculations.

```
    function getSwapAmountForAddLiquidity(
        uint256 tokenAmount0,
        uint256 tokenAmount1
    ) internal view returns (int256 amount0, int256 amount1) {
        VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
            .getVaultStorage();
        VaultTypes.VeloVaultState storage vaultState = vaultStorage.state;
        (uint256 reserve0, uint256 reserve1, ) = IPairV2(vaultState.pair)
            .getReserves();

        uint amountOptimal;
        uint swapAmount;
        uint excessAmount;

        // loop to find the optimal amount
        for (int i = 0; i < 3; i++) {
            if (tokenAmount0.mul(reserve1) > tokenAmount1.mul(reserve0)) {
                amountOptimal = tokenAmount1.mul(reserve0).div(reserve1);
                excessAmount = tokenAmount0.sub(amountOptimal);

                // swap excessAmount/2 token0 to token1
                swapAmount = excessAmount.div(2);
                if (swapAmount <= vaultState.minSwapAmount0) {
                    break;
                }

                amount0 = amount0 - int256(swapAmount);
                tokenAmount0 = swapAmount;
                tokenAmount1 = IPairV2(vaultState.pair).getAmountOut(
                    swapAmount,
                    vaultState.token0
```

```
                );
                amount1 = amount1 + int256(tokenAmount1);

                reserve0 = reserve0.add(swapAmount);
                reserve1 = reserve1.sub(tokenAmount1);
            } else {
                amountOptimal = tokenAmount0.mul(reserve1).div(reserve0);
                excessAmount = tokenAmount1.sub(amountOptimal);
                // swap excessAmount/2 token1 to token0
                swapAmount = excessAmount.div(2);
                if (swapAmount <= vaultState.minSwapAmount1) {
                    break;
                }

                amount1 = amount1 - int256(swapAmount);
                tokenAmount1 = excessAmount.sub(swapAmount);
                tokenAmount0 = IPairV2(vaultState.pair).getAmountOut(
                    swapAmount,
                    vaultState.token1
                );
                amount0 = amount0 + int256(tokenAmount0);

                reserve1 = reserve1.add(swapAmount);
                reserve0 = reserve0.sub(tokenAmount0);
            }
        }
    }
```

Listing 2.11: VeloLiquidityMath.sol

The `getSwapAmountForExactOut` function in the `VeloLiquidityMath` library calculates the required `amountIn` to achieve a specific `exactOut` amount. Initially, it estimates an approximate `amountIn` using the `getAmountOut` function with `exactOut` as the input. It then incrementally increases `amountIn` by 1% up to 10 times to determine if the resulting `amountOut` reaches the `exactOut`.

However, this function may produce inaccurate results due to the following reasons:

- First, if the initially estimated `amountIn` already exceeds `exactOut`, the function returns `amountIn` as the final result without any further 1% increases.
- Second, if the calculated `amountOut` still does not reach `exactOut` after 10 iterations, the returned `amountIn` is not accurate.

```
    function getSwapAmountForExactOut(
        address from,
        address to,
        uint256 exactOut
    ) internal view returns (uint256) {
        VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
            .getVaultStorage();
        VaultTypes.VeloVaultState storage vaultState = vaultStorage.state;

        uint256 amountIn = IPairV2(vaultState.pair).getAmountOut(exactOut, to);

        for (uint256 i = 0; i < 10; i++) {
```

```
            amountIn = (amountIn * 1010) / 1000;

            uint256 out = IPairV2(vaultState.pair).getAmountOut(amountIn, from);

            if (out >= exactOut) {
                break;
            }
        }

        return amountIn;
    }
```

Listing 2.12: VeloLiquidityMath.sol

Feedback from the Developers There may indeed be some inaccuracies in the results, but the level of inaccuracy here does not affect asset security.

- The `getSwapAmountForAddLiquidity` method is primarily used to calculate the optimal swap amount to maximize liquidity provision when opening a position. If the calculation result is not accurate, it may result in a small amount of tokens not being provided as liquidity but remaining in the user's position. However, when the user closes the position, they can still retrieve this small portion of tokens.
- The `getSwapAmountForExactOut` method is used to calculate the sufficient amount of tokens to repay the debt when closing a position. Therefore, as long as there is enough, it is acceptable. If there are slight inaccuracies, any excess tokens will still be returned to the user's wallet when closing the position.