Security Audit Report for Whale.Loans Contracts

Date: Nov 25, 2021

Version: 1.4

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Whale.Loans Ltd.
Target	Whale.Loans Contracts

Version History

Version	Date	Description
1.0	Nov 8, 2021	First Release
1.1	Nov 12, 2021	Second Release
1.2	Nov 16, 2021	Third Release
1.3	Nov 23, 2021	Remove unnecessary concerns
1.4	Nov 25, 2021	Update for the new repo

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

The target contract is Whale.Loans Contracts. The detailed description is in the following link: Whale.Loans.

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

The files that are audited in this report include the following ones.

Repo Name	Github URL
Whale.loans smart contracts	https://github.com/Whale-loans/contracts

The commit hash before the audit is `e3ca48812f586d351d18116ef127982d3f21e493`. The commit hash that fixes the issues found in this audit is `53438be9a30cb1c88e4fd40ebb5e84980564136f`.

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 do 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 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
- Access control
- 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 [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

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

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

Chapter 2 Findings

In total, we find five potential issues, two recommendations in Whale.Loans Contracts, as follows:

- High Risk: 2
- Medium Risk: 3
- Recommendation: 2

ID	Severity	Description	Category
1	High	*Reentrancy Vulnerability in FlashERC20*	Software Security
2	High	`stakedAmount` *Manipulation in YieldDistribution*	DeFi Security
3	Medium	*Improper Implementation of Reward Mechanism in* `recomputeFees()`	DeFi Security
4	Medium	*Unclaimed Fees (*`stakedAmount`*) Overriding*	DeFi Security
5	Medium	*Improper Implementation of* `executeWithdraw()`	DeFi Security
6	-	*Removing Redundant* `require()` *Statement*	Recommendation
7	-	*Do Not Use Elastic Supply Tokens*	Recommendation

The details are provided in the following sections.

2.1 Software Security

2.1.1 Reentrancy Vulnerability in FlashERC20

Status Confirmed and fixed.

Description Users can deposit their underlying tokens (through `FlashERC20.deposit()`) to FlashERC20 as staked tokens, and harvest "fees" as rewards (through `FlashERC20.claimFees()`) afterwards. However, there exists a potential reentrancy vulnerability in FlashERC20 with `deposit()` and `claimFees()`.

```
function deposit(uint256 wad) public {
    underlying.safeTransferFrom(msg.sender, address(this), wad);
    _mint(msg.sender, wad);
    yieldDistribution.computeDeposit(wad, msg.sender);
    assert(underlying.balanceOf(address(this)) <= _depositLimit);
    emit Deposit(msg.sender, wad);
}
```

Listing 2.1: deposit:FlashERC20.sol

```
function claimFees(uint256 amount) public {
    uint256 value = yieldDistribution.claimFees(msg.sender, amount);
    underlying.safeTransfer(msg.sender, value);
}
```

Listing 2.2: claimFees:FlashERC20.sol

```
function claimFees(address stakeholder, uint256 amount)
    public
    returns (uint256)
{
    require(crops[msg.sender].authorised, "!authorised");
    require(!isUpdatable(msg.sender), "!recomputeFees");

    syncStakeholder(stakeholder);

    require(
        crops[msg.sender].claimableTotalGrowthSync >=
            getClaimablePeriod(msg.sender),
        "Claimable total growth is not up to date"
    );
    (uint256 positiveFees, uint256 negativeFees) = getFees(
        msg.sender,
        stakeholder,
        0
    );
...
```

Listing 2.3: claimFees:YieldDistribution.sol

```
function getFees(
    address yg,
    address stakeholder,
    uint256 precomputedClaimableTotalGrowth
) public view returns (uint256 positive, uint256 negative) {
    uint256 claimableTotalGrowth;
    uint256 claimablePeriod = getClaimablePeriod(yg);

    if (crops[yg].claimableTotalGrowthSync >= claimablePeriod) {
        claimableTotalGrowth = crops[yg].claimableTotalGrowth;
    } else {
        // If we enter here it means that we are calling this function from outside the app
        // so the following calculation will never occur during a fees claim as it would
        // be expensive in terms of gas
        claimableTotalGrowth = precomputedClaimableTotalGrowth != 0
            ? precomputedClaimableTotalGrowth
            : getClaimableTotalGrowth(yg);
    }

    if (claimableTotalGrowth == 0) return (0, 0);
    AlreadyClaimed memory alreadyClaimed = crops[yg].claimedAmount[
        stakeholder
    ];
```

```
        AlreadyClaimed memory totalClaims = crops[yg].totalClaims;

        (uint256 positiveTotalFees, uint256 negativeTotalFees) = YieldGenerator(
            yg
        ).getTotalFees();

        if (negativeTotalFees != 0) {
            negative = calculateFee(
                yg,
                stakeholder,
                negativeTotalFees,
                alreadyClaimed.negative,
                claimableTotalGrowth,
                totalClaims.negative
            );
        } else if (positiveTotalFees != 0) {
            positive = calculateFee(
                yg,
                stakeholder,
                positiveTotalFees,
                alreadyClaimed.positive,
                claimableTotalGrowth,
                totalClaims.positive
            );
        }
    }
```

Listing 2.4: getFees:YieldDistribution.sol

```
    function calculateFee(
        address yg,
        address _stakeholder,
        uint256 _totalFees,
        uint256 _alreadyClaimed,
        uint256 _cTotalGrowth,
        uint256 _totalClaims
    ) internal view returns (uint256) {
        return
            getClaimableGrowth(yg, _stakeholder)
                .mul(_totalFees.add(_totalClaims))
                .div(_cTotalGrowth)
                .sub(_alreadyClaimed);
    }
```

Listing 2.5: calculateFee:YieldDistribution.sol

When a user wants to claim fees (rewards), `YieldDistribution.calculateFee()` is invoked to calculate the current maximum claimable fees for that user ($userClaimableFees$), and the formula is as follows:

$$\begin{aligned} userClaimableFees = & \frac{userClaimableGrowth}{totalClaimableGrowth} \times (totalFees + totalClaims) \\ & - userAlreadyClaimed \end{aligned} \tag{2.1}$$

$$totalFees = underlying.balanceOf(fERC20) - fERC20.totalSupply() \quad (2.2)$$

As $userClaimableFees$ increases with the increase of $totalFees$, an attacker could launch an reentrancy attack by manipulating $totalFees$ to increase $userClaimableFees$ **under the condition that the underlying token supports callback mechanism and the callback function is executed after token transfer**, and finally claim more fees. The exploitation steps are as follows:

1. Suppose an attacker has previously deposited (in previous periods).
2. Later, the attacker calls `FlashERC20.deposit()` that further invokes `underlying.safeTransferFrom()`, in which the underlying tokens are transferred to FlashERC20 followed by executing attacker's callback function. **NOTICE: underlying tokens have been transferred to FlashERC20 before executing attacker's callback function, this is the key of the exploitation**.
3. In the callback function, the attacker calls `FlashERC20.claimFees()` (FlashERC20 conntract reentrancy), in which $userClaimableFees$ should be calculated firstly. At this time the underlying tokens have been transferred to FlashERC20 while `FlashERC20._mint()` has not been invoked, so $underlying.balanceOf(fERC20)$ increases but $fERC20.totalSupply()$ remains the same. According to equations 2.2 and 2.1, $totalFees$ and $userClaimableFees$ both increase.
4. Finally, the attacker could harvest more fees.

Impact The attacker could attack the system to steal underlying tokens from FlashERC20. What's worse, the loss could be magnified by using flashloan.

Suggestion Add sanity checks to prevent the reentrancy attack.

2.2 DeFi Security

2.2.1 `stakedAmount` Manipulation in YieldDistribution

Status Confirmed and fixed.

Description `stakedAmount` in YieldDistribution could be maliciously manipulated in the following two ways:

1. **Flashloan Attack**

   An attacker may launch the flashloan attack by manipulating `stakedAmount`, as follows:

   1) invoking the `flashMint()` function in FlashERC20 or FlashMain to borrow a large amount of `fERC20`, and the execution would go back to the attacker's `executeOnFlashMint()`.
   2) invoking the `FlashERC20.withdraw()` function to redeem the underlying tokens.
   3) invoking the `FlashERC20.deposit()` function, which further calls the `computeDeposit()` function of YieldDistribution. By doing so, the deposit amount will be added to the attacker's `stakedAmount`, which contributes to the reward fees of the economic system. Namely, the attacker "stakes" what he borrows from flashMint.
   4) repaying the `fERC20` borrowed from the `flashMint()`.
   5) finally, claiming the fees retrieved in step 3, and no cost is taken in the whole process.

2. **"Double Spending" Attack**

   An attacker may also launch the "double spending" attack by manipulating the `stakedAmount` in the following steps:

1) depositing some underlying tokens to the FlashERC20 by using account A. YieldDistribution then records the `stakedAmount` of User A.
2) transferring `fERC20` to another account B, which can withdraw `fERC20` to redeem underlying tokens from FlashERC20.

At this moment, the attacker could double (event triple or more, if more accounts are utilized) `stakedAmount` and then collect fees without any risk.

```
function withdraw(uint256 wad) public {
    _burn(msg.sender, wad); // reverts if `msg.sender` does not have enough fERC20
    underlying.safeTransfer(msg.sender, wad);
    yieldDistribution.computeWithdraw(wad, msg.sender);
    emit Withdrawal(msg.sender, wad);
}
```

Listing 2.6: withdraw:FlashERC20.sol

```
function computeWithdraw(uint256 amount, address recipient)
    public
    returns (uint256)
{
    require(crops[msg.sender].authorised, "!authorised");
    require(!isUpdatable(msg.sender), "!recomputeFees");

    if (crops[msg.sender].stakedAmount[recipient] <= amount) {
        crops[msg.sender].stakedAmount[recipient] = 0;
        deleteStakeHolder(recipient);
    } else {
        crops[msg.sender].stakedAmount[recipient] = crops[msg.sender]
            .stakedAmount[recipient]
            .sub(amount);
        syncStakeholder(recipient);
    }
    return crops[msg.sender].stakedAmount[recipient];
}
```

Listing 2.7: computeWithdraw:YieldDistribution.sol

```
function deposit(uint256 wad) public {
    underlying.safeTransferFrom(msg.sender, address(this), wad);
    _mint(msg.sender, wad);
    yieldDistribution.computeDeposit(wad, msg.sender);
    assert(underlying.balanceOf(address(this)) <= _depositLimit);
    emit Deposit(msg.sender, wad);
}
```

Listing 2.8: deposit:FlashERC20.sol

```
function computeDeposit(uint256 amount, address recipient)
    public
    returns (uint256)
{
    require(crops[msg.sender].authorised, "!authorised");
    require(!isUpdatable(msg.sender), "!recomputeFees");
```

```

        syncStakeholder(recipient);

        uint256 next = getNextFeeDistribution(msg.sender);

        if (crops[msg.sender].stakedAmount[recipient] == 0) {
            if (next.sub(block.timestamp) < crops[msg.sender].minTime) {
                crops[msg.sender].stakeDate[recipient] = next;
            } else {
                crops[msg.sender].stakeDate[recipient] = block.timestamp;
            }
        } else {
            uint256 effectiveTimestamp;
            if (next.sub(block.timestamp) < crops[msg.sender].minTime) {
                effectiveTimestamp = next;
            } else {
                effectiveTimestamp = block.timestamp;
            }
            crops[msg.sender].stakeDate[recipient] = amount
                .mul(effectiveTimestamp)
                .add(
                    crops[msg.sender].stakedAmount[recipient].mul(
                        crops[msg.sender].stakeDate[recipient]
                    )
                )
                .div(crops[msg.sender].stakedAmount[recipient].add(amount));
        }
        crops[msg.sender].stakedAmount[recipient] = crops[msg.sender]
            .stakedAmount[recipient]
            .add(amount);
        crops[msg.sender].stakeHolders.add(recipient);

        return crops[msg.sender].stakedAmount[recipient];
    }
```

Listing 2.9: computeDeposit:YieldDistribution.sol

Impact The attacker could attack the system to **constantly and infinitely** get the system's reward with **no risk, and almost no cost**.

Suggestion Ensure the consistency of `fERC20`, underlying tokens and `stakedAmount`.

2.2.2 Improper Implementation of Reward Mechanism in `recomputeFees()`

Status Confirmed and fixed.

Description The `FlashERC20.recomputeFees()` function calls `YieldDistribution.recomputeFees()` followed by `computeDeposit()` (when `_updateReward` is not zero). However, the `recomputeFees()` function only updates the growth of stakeholders in one batch, and the `computeDeposit()` function requires that the yield generator is fully updated. As a result, if there are more than one non-updated batch (default 100) of stakeholders and `_updateReward` is not 0, any call to `FlashERC20.recomputeFees()` will fail.

```
    function recomputeFees() external {
```

```
        yieldDistribution.recomputeFees(address(this));

        if (_updateReward != 0) {
            yieldDistribution.computeDeposit(
                _updateReward,
                isOwner() ? address(this) : msg.sender
            );
        }
    }
```

Listing 2.10: recomputeFees:FlashERC20.sol

```
    function recomputeFees(address yg) external {
        require(crops[yg].authorised, "!authorised");
        require(isUpdatable(yg), "!isUpdatable");
        (
            uint256 ctg,
            uint256 lastStakeholderIdx,
            bool isLastBatch
        ) = getClaimableTotalGrowthBatch(
                yg,
                crops[yg].lastStakeholderSyncedIdx
            );
        if (crops[yg].lastStakeholderSyncedIdx == 0) {
            crops[yg].claimableTotalGrowth = 0;
        }
        if (isLastBatch) {
            crops[yg].claimableTotalGrowthSync = block.timestamp;
            crops[yg].lastStakeholderSyncedIdx = 0;
        } else {
            crops[yg].lastStakeholderSyncedIdx = lastStakeholderIdx;
        }
        crops[yg].claimableTotalGrowth += ctg;
    }
```

Listing 2.11: recomputeFees:YieldDistribution.sol

```
    function computeDeposit(uint256 amount, address recipient)
        public
        returns (uint256)
    {
        require(crops[msg.sender].authorised, "!authorised");
        require(!isUpdatable(msg.sender), "!recomputeFees");
    ......
```

Listing 2.12: computeDeposit:YieldDistribution.sol

Impact The mechanism of `recomputeFees()` will fail and the stakers may lose their rewards.

Suggestion Re-design the `computeDeposit()` implementation to be compatible with `recomputeFees()`.

2.2.3 Unclaimed Fees (`stakedAmount`) Overriding

Status Confirmed and fixed.

Description The `YieldDistribution.computeWithdraw()` function does not consider accumulated and unclaimed fees of stakeholders. For example, if a stakeholder has never called `FlashERC20.claimFees()` since her first deposit, then the invocation of `FlashERC20.withdraw()` (which calls the `computeWithdraw()` of YieldDistribution) will clear part of (or all) unclaimed fees of the stakeholder.

```
function withdraw(uint256 wad) public {
    _burn(msg.sender, wad); // reverts if ‘msg.sender‘ does not have enough fERC20
    underlying.safeTransfer(msg.sender, wad);
    yieldDistribution.computeWithdraw(wad, msg.sender);
    emit Withdrawal(msg.sender, wad);
}
```

Listing 2.13: withdraw:FlashERC20.sol

```
function computeWithdraw(uint256 amount, address recipient)
    public
    returns (uint256)
{
    require(crops[msg.sender].authorised, "!authorised");
    require(!isUpdatable(msg.sender), "!recomputeFees");

    if (crops[msg.sender].stakedAmount[recipient] <= amount) {
        crops[msg.sender].stakedAmount[recipient] = 0;
        deleteStakeHolder(recipient);
    } else {
        crops[msg.sender].stakedAmount[recipient] = crops[msg.sender]
            .stakedAmount[recipient]
            .sub(amount);
        syncStakeholder(recipient);
    }
    return crops[msg.sender].stakedAmount[recipient];
}
}
```

Listing 2.14: computeWithdraw:YieldDistribution.sol

Impact Stakeholders may lose their rewards when calling `withdraw()` without claiming fees.

Suggestion Re-implement the `withdraw()` and `computeWithdraw()` to deal with unclaimed fees.

2.2.4 Improper Implementation of `executeWithdraw()` in DeltaNeutralVault

Status Confirmed and fixed.

Description The `executeWithdraw()` function in DeltaNeutralVault is implemented to first fetch the amount of tokens stored in *Kitten*'s vault (i.e., `vault`), and then execute different handling logic by comparing `vault` and `amt` (i.e., the requested withdrawal amount). Specifically, if `vault` is less than `amt`, first swapping from bull/bear to Kitten's vault (the amount is determined by `amt`) and then calling `kWithdraw()` (which withdraws tokens from Kitten's vault to DeltaNeutralVault). However, if `vault` is *NOT* less than than `amt`, no action will be performed (i.e., `kWithdraw()` will not be executed), hence the withdrawal from Kitten's vault to DeltaNeutralVault would fail.

```
function executeWithdraw(uint256 amt) internal returns (uint256) {
```

```
    KAddresses memory addresses = KAddresses(
        address(K),
        gFeed,
        address(underlying),
        address(this)
    );
    (uint256 bullValue, uint256 bearValue, uint256 vault, ) = dnTools
        .kGetTotalBalance(addresses);

    if (vault < amt) {
        uint256 initialBalance = underlying.balanceOf(address(this));

        bool order = bullValue > bearValue;

        if ((order ? bullValue : bearValue) >= amt.sub(vault)) {
            kSwap(
                order ? 1 : 2,
                0,
                dnTools.kToVault(addresses, amt.sub(vault), order ? 1 : 2)
            );
        } else {
            kSwap(
                order ? 1 : 2,
                0,
                dnTools.kToVault(
                    addresses,
                    order ? bullValue : bearValue,
                    order ? 1 : 2
                )
            );
            kSwap(
                order ? 2 : 1,
                0,
                dnTools.kToVault(
                    addresses,
                    order ? bearValue : bullValue,
                    order ? 2 : 1
                )
            );
        }

        kWithdraw(amt);

        return underlying.balanceOf(address(this)).sub(initialBalance);
    }
    return amt;
}
```

Listing 2.15: executeWithdraw:DeltaNeutralKitten.sol

Impact When $vault \geq amt$, `executeWithdraw()` does not work correctly, i.e., withdrawal action will fail.

Suggestion Ensure that `kWithdraw()` is always invoked in `executeWithdraw()`.

2.3 Additional Recommendation

2.3.1 Removing Redundant `require()` Statement

Status Confirmed and fixed.

Description The `require()` statements in line 349 and line 353 $\sim$ 357 in `YieldDistribution.claimFees()` have the same effect.

```
344    function claimFees(address stakeholder, uint256 amount)
345        public
346        returns (uint256)
347    {
348        require(crops[msg.sender].authorised, "!authorised");
349        require(!isUpdatable(msg.sender), "!recomputeFees");
350
351        syncStakeholder(stakeholder);
352
353        require(
354            crops[msg.sender].claimableTotalGrowthSync >=
355                getClaimablePeriod(msg.sender),
356            "Claimable total growth is not up to date"
357        );
358    ...
```

Listing 2.16: claimFees:YieldDistribution.sol

Impact A waste of gas.

Suggestion Remove either one `require` statement.

2.3.2 Do Not Use Elastic Supply Tokens

Status Unknown

Description & Suggestion Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. Such as inflationary token, deflationary token, rebasing token, and so forth. Such a mechanism makes a DeFi system over complex. For example, a DEX using deflationary token must double check the token transfer amount when taking swap action because of the difference of actual transfer amount and parameter. The abuse of elastic supply tokens will make the DeFi system vulnerable. In reality, many security accidents are caused by the elastic supply tokens. In terms of confidentiality, integrity and availability, we highly recommend that do not use elastic supply tokens.

Impact N/A

Suggestion N/A

Chapter 3 Conclusion

In this audit, we have analyzed the business logic, the design, and the implementation of the Whale.Loans Contracts. Indeed, we are impressed by the design of Whale.Loans Contracts that tries to provide flash minting services (and Delta Neutral Vaults) with a decentralized solution. Overall, the current code base is well structured and implemented.

Meanwhile, as previously disclaimed, this report does not give any warranties on discovering all security issues of the smart contracts. We appreciate any constructive feedback or suggestions.