Blur - Blend Security Audit

: Blend, NFT Collateralized ETH Lending Platform

April 28, 2023

Revision 1.1

ChainLight@Theori

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.

Table of Contents

Blur - Blend Security Audit 1
Table of Contents 2
Executive Summary 3
Audit Overview 4
Scope 4
Code revision 4
Severity Categories 5
Status Categories 6
Finding Breakdown by Severity 7
Findings 8
Summary 8
#1 BLEND-001 Payable wrapper functions in the Blend contract do not credit the BlurPool deposit to the user 10
#2 BLEND-002 An attacker can consume all loan offers for a collection with just one NFT 11
#3 BLEND-003 Functions should validate Lien.auctionDuration on changes 13
#4 BLEND-004 A malicious lender can quickly set a lien to a defaulted state when the loan offer had small auctionDuration value 14
#5 BLEND-005 An order created in buyToBorrow() and takeBid() can have a hash identical to the previously executed one 16
#6 BLEND-006 _buyLocked() emits BuyLocked event with incorrect arguments 17
#7 BLEND-007 Potential read-only reentrancy vector in buyLocked(), repay(), and takeBid() 18
#8 BLEND-008 Blend.computeCurrentDebt() can return a much higher interest rate limit than _LIQUIDATION_THRESHOLD 20
#9 BLEND-009 Use two-step ownership transfer 22
Revision History 23

Executive Summary

Starting on April 10th, 2023, ChainLight of Theori audited the smart contract of Blur's Blend for two weeks. Blend is an NFT collateralized ETH lending platform that provides a range of features, including borrowing, repayment, refinancing, Dutch auctions, and marketplace interactions, which empower users to sell collateral for loan repayment or finance NFT purchases with loans.

Since the contract transfers NFTs and customized wrapped ETH on behalf of users to settle trades/loans, we focused on identifying issues that may allow an attacker to trigger relevant features while impersonating another user and whether the collateral can be withdrawn without repayment of a loan, etc. In addition, We meticulously examined the functions that accept a loan offer struct and a lien struct as arguments, ensuring that the loan offer's signature is always validated and that the lien's hash is always checked if it equals the one stored in the contract. Such checks are crucial to avoid severe consequences like theft of funds by an attacker providing arbitrarily crafted data.

As a result, we identified two high-severity issues, including an issue that may lead to the lock of a user's funds and an issue that may lead to excessive interest rate, and also a medium-severity issue that may lead to a griefing attack, four low-severity issues, and two informational issues.

Audit Overview

Scope

Name	Blur - Blend Security Audit
Target / Version	• Git Repository (blend-audit): commit `ab203fe02cc2d7692b7cf933d1c57d3949717f31` (`audit` branch)
Application Type	Smart contracts
Lang. / Platforms	Smart contracts [Solidity]

Code revision

N/A

Severity Categories

Severity	Description
Critical	The attack cost is low (not requiring much time or effort to succeed in the actual attack), and the vulnerability causes a high-impact issue. (e.g., Effect on service availability, Attacker taking financial gain)
High	An attacker can succeed in an attack which clearly causes problems in the service's operation. Even when the attack cost is high, the severity of the issue is considered "high" if the impact of the attack is remarkably high.
Medium	An attacker may perform an unintended action in the service, and the action may impact service operation. However, there are some restrictions for the actual attack to succeed.
Low	An attacker can perform an unintended action in the service, but the action does not cause significant impact or the success rate of the attack is remarkably low.
Informational	Any informational findings that do not directly impact the user or the protocol.

Status Categories

Status	Description
Confirm	ChainLight reported the issue to the vendor, and they confirm that they received.
Reported	ChainLight reported the issue to the vendor.
Fixed	The vendor resolved the issue.
Acknowledged	The vendor acknowledged the potential risk, but they will resolve it later.
WIP	The vendor is working on the patch.
Won't Fix	The vendor acknowledged the potential risk, but they decided to accept the risk.

Finding Breakdown by Severity

Category	Count	Findings
Critical	0	• N/A
High	2	• `BLEND-001` • `BLEND-008`
Medium	1	• `BLEND-002`
Low	4	• `BLEND-003` • `BLEND-004` • `BLEND-005` • `BLEND-006`
Informational	2	• `BLEND-007` • `BLEND-009`

Findings

Summary

#	ID	Title	Severity	Status
1	`BLEND-001`	Payable wrapper functions in the `Blend` contract do not credit the `BlurPool` deposit to the user	High	Fixed
2	`BLEND-002`	An attacker can consume all loan offers for a collection with just one NFT	Medium	Fixed
3	`BLEND-003`	Functions should validate `Lien.auctionDuration` on changes	Low	Fixed
4	`BLEND-004`	A malicious lender can quickly set a lien to a defaulted state when the loan offer had small `auctionDuration` value	Low	Won't Fix
5	`BLEND-005`	An order created in `buyToBorrow()` and `takeBid()` can have a hash identical to the previously executed one	Low	Fixed
6	`BLEND-006`	`_buyLocked()` emits `BuyLocked` event with incorrect arguments	Low	Fixed
7	`BLEND-007`	Potential read-only reentrancy vector in `buyLocked()`, `repay()`, and `takeBid()`	Informational	Acknowledged

#	ID	Title	Severity	Status
8	`BLEND-008`	`Blend.computeCurrentDebt()` can return a much higher interest rate limit than `_LIQUIDATION_THRESHOLD`	High	Fixed
9	`BLEND-009`	Use two-step ownership transfer	Informational	Fixed

#1 `BLEND-001` Payable wrapper functions in the `Blend` contract do not credit the `BlurPool` deposit to the user

ID	Summary	Severity
`BLEND-001`	Payable wrapper functions in `Blend` using `safeTransferETH()` does not increase the `BlurPool` balance of the `msg.sender`, but increase the balance of the `Blend` contract instead.	High

Description

Payable wrapper functions in `Blend` contract such as `buyToBorrowETH`, `buyToBorrowLockedETH`, and `buyLockedETH` uses `safeTransferETH()`.

`BlurPool.deposit()` credits the received ETH to the `msg.sender`, which is not the intended account in this case since the `msg.sender` of an internal transaction to the `BlurPool` contract will be the `Blend` contract rather than the user.

Impact

High

The transaction will fail, or if a user has enough balance in the `BlurPool` to execute the trade, funds will be frozen until it is rescued via an upgrade of the `BlurPool` contract.

Recommendation

`BlurPool` balance should be credited to the right user. Adding `pool.transferFrom(address(this), msg.sender, msg.value)` after `safeTransferETH()` in the payable wrapper functions would do it.

Patch

Fixed

Deposit-on-behalf function is added to the `BlurPool` contract and used by payable wrapper functions.

#2 `BLEND-002` An attacker can consume all loan offers for a collection with just one NFT

ID	Summary	Severity
`BLEND-002`	It is possible to call `repay()` within the same block immediately after a user takes a loan, which does not incur a fee or interest. As a result, a malicious borrower can execute a griefing attack consuming all loan offers for a collection by repeatedly calling `borrow()` and `repay()` while holding just one NFT in the collection.	Medium

Description

The main problem is that the `repay()` does not require a minimum loan duration or minimum fee in the current Blend system. Therefore, a malicious borrower can consume all loan offers for a collection by repeatedly calling `borrow()` and `repay()` in the same block. In this scenario, the borrower only spends gas since they only need to return the loan's principal without any interest. But the lender has to sign a new loan offer again, which leads to a bad user experience.

Additionally, the borrower can use Blend system as a fee-free flash loan provider. Let us assume there is a malicious borrower contract. The `repay` function calls `safeTransferFrom()` for NFT transfer before taking the balance of the borrower (`pool.transferFrom(msg.sender, lien.lender, debt);`). Since `safeTransferFrom()` internally calls the `onERC721Received()` of the `to` contract after the transfer, the borrower can call `borrow()` with another loan offer of the same collection in the `onERC721Received` function. (`borrow(loanOfferA) -> repay(loanOfferA) -> onERC721Received() -> borrow(loanOfferB) -> do something -> pool.deposit -> repay(loanOfferB)`) This can be nested to borrow more as much as the call stack depth limit of EVM allows. In such cases, lenders do not receive any compensation for providing liquidity while suffering the inconvenience of signing the new loan offers again.

Impact

Medium

1. All loan offers for a collection can be consumed in one or a few transactions, so lenders have to create and sign a new `LoanOffer`.
2. A borrower can execute a flash loan using the lender's `LoanOffer`, while no fees are paid to the lenders.
3. Refinancing can be hindered by repeatedly wiping out loan offers, which may lead to defaults.

Recommendation

1. Introduce a minimum fee or minimum loan duration for the `repay` function.
2. Implement the auto-renewal of loan offers by decreasing the `amountTaken` once the debt is repaid.

Patch

Fixed

A floor for loan duration is introduced for interest calculation. (Recommendation #1)

#3 `BLEND-003` Functions should validate `Lien.auctionDuration` on changes

ID	Summary	Severity
`BLEND-003`	`buyToBorrow()` and `borrowerRefinance()` do not check if `auctionDuration` is less than or equal to `_MAX_AUCTION_DURATION`.	Low

Description

According to the docs, `auctionDuration` can not get over `_MAX_AUCTION_DURATION`.

The `borrow()`, `refinance()`, and `refinanceAuctionByOther()` validates `auctionDuration`, but the `buyToBorrow()` and `borrowerRefinance()` does not.

Impact

Low

A user's mistake or a bug in the frontend may lead to a lien with bad auction parameters. For example, a `lender` can set a huge `auctionDuration` by mistake and may not be able to call `seize()` for that lien forever.

Recommendation

Missing checks for the `auctionDuration` should be added to `buyToBorrow()`, and `borrowerRefinance()`.

Patch

Fixed

It is fixed as recommended.

#4 `BLEND-004` A malicious lender can quickly set a lien to a defaulted state when the loan offer had small `auctionDuration` value

ID	Summary	Severity
`BLEND-004`	A lender can call `startAuction()` at any point after `borrow()` is called. So a malicious lender can quickly set a lien to the defaulted state when the loan offer had a very short `auctionDuration`.	Low

Description

The main problem is that the loan duration is not deterministic when the `borrow()` is executed. Since a lender can call `startAuction()` at any time, the borrower can't estimate when it will be triggered, although they can check the auction duration.

If `auctionDuration` is low enough, even if it is not 0, it is possible that the loan is not refinanced timely by `refinanceAuctionByOther()` or `refinanceAuction()`. To protect their locked NFT in such cases, borrowers would need to monitor the call to `startAuction()`, but it would be infeasible for most non-technical users.

For a worse case, let's assume the `acutionDuration` is 0 and the borrower was mistaken or tricked into taking that `LoanOffer`, then the lender can immediately call `startAuction()` in the same block. In the next block, the lien is considered as defaulted (`_lienIsDefaulted()` returns true). The lender can effectively buy an NFT at a significant discount by calling the `seize()` function to give up the loaned amount and get the locked NFT instead.

Impact

Low

If a borrower accepts a `LoanOffer` with a low `auctionDuration` value, the lien may change to the defaulted state in a very short time, causing the locked NFT to be seized. Users must review the terms before taking the loan, but allowing such errors may result in a bad user experience.

Recommendation

1. Add the `auctionDelay` variable to the `LoanOffer` struct. And add a check preventing the lender from calling `startAuction()` until an additional `auctionDelay` seconds has elapsed after the `LoanOffer.startTime`.

2. Add a minimum threshold for the `auctionDuration`

Patch

Won't Fix

Blur team decided not to address the issue because the borrower is agreeing to the `auctionDuration` when they take the loan offer and thus consent to the potentially quick default.

#5 `BLEND-005` An order created in `buyToBorrow()` and `takeBid()` can have a hash identical to the previously executed one

ID	Summary	Severity
`BLEND-005`	Although Blur exchange contract does not allow the same order hash to be reused, the new order that has a hash identical to the previously executed order can be created in the `buyToBorrow()` and `takeBid()`	Low

Description

The `buyToBorrow()` and `takeBid()` call `execute()` of Blur exchange contract for buying/selling NFT. In the `execute()`, the order hash is marked to prevent the re-execution of a filled or canceled order. Nonetheless, since `salt` is set to zero for Blend's orders, an identical taker order would be created when the `tokenId` is the same and `execution.makerOrder` has the same direction, `price` and `listingTime` as the previously filled order. (The same `price` and `listingTime` can exist among multiple asks/bids for one NFT collection.)

Impact

Low

If a maker order is filled for a specific NFT, other maker orders with the same direction, `price` and `listingTime` cannot be used for that NFT in `buyToBorrow()` and `takeBid()` forever.

Recommendation

Specify the proper salt (e.,g. `lienId`) rather than zero when creating an order in `buyToBorrow()` and `takeBid()`.

Patch

Fixed

`lienId` is used as salt in `takeBid()`, and `execution.makerOrder.order.trader` is used as salt in `buyToBorrow()`.

#6 `BLEND-006` `_buyLocked()` emits `BuyLocked` event with incorrect arguments

ID	Summary	Severity
`BLEND-006`	`BuyLocked` event is declared with arguments `lienId`, `collection`, `buyer`, `seller`, and `tokenId`. However, the event is emitted with `lienId`, `buyer`, `seller`, `collection`, and `tokenId` in the `_buyLocked()` function.	Low

Description

In `_buyLocked()` function, `BuyLocked` event is emitted with `offer.lienId`, `msg.sender`, `lien.borrower`, `address(lien.collection)` and `lien.tokenId` arguments, but this order does not match the order of the arguments in `BuyLocked` event declaration.

Impact

Low

Codes referencing `BuyLocked` event may malfunction.

Recommendation

Change the order of event arguments in either the event declaration or the event emission.

Patch

Fixed

It is fixed as recommended.

#7 `BLEND-007` Potential read-only reentrancy vector in `buyLocked()`, `repay()`, and `takeBid()`

ID	Summary	Severity
`BLEND-007`	Integrations with `Blend` contract may be vulnerable to read-only reentrancy due to external calls in `buyLocked()`, `repay()`, and `takeBid()`.	Informational

Description

`buyLocked()`, `repay()`, and `takeBid()` in `Blend` contract may trigger a call to an attacker-controlled contract via ERC-721's `onERC721Received` callback feature, while some states (balances in `BlurPool`) are not updated yet. Therefore, integrations with `Blend` contract that tries to use `pool.balanceOf()` may be vulnerable to read-only reentrancy depending on their implementation.

Also, if a feature that allows a temporary increase of the `pool.balanceOf(blend)` is added (like deposit/withdraw, stake/unstake, mint/redeem), it may enable theft of funds held by blend contract; however, it doesn't seem likely at this point.

Impact

Informational

Although the impact is negligible for the current version, it should be re-evaluated when there is any code modification.

Recommendation

Add a reentrancy guard and expose the guard's state.

Reentrancy in `buyLocked()` and `repay()` can be eliminated by moving the `collection.safeTransferFrom` call to the right before the end of the function. However, it would be difficult to do the same for `takeBid()` since `safeTransferFrom` is called inside the call to `Blur.execute()`.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3829

Patch

Acknowledged

Blur team decided not to address the issue since no active exploit exists. But they will resolve the issue if there are ever upgrades that may create a vulnerability.

#8 `BLEND-008` `Blend.computeCurrentDebt()` can return a much higher interest rate limit than `_LIQUIDATION_THRESHOLD`

ID	Summary	Severity
`BLEND-008`	`Blend.computeCurrentDebt()` can return an interest rate limit greater than `_LIQUIDATION_THRESHOLD` because of an error in the calculation formula.	High

Description

`Blend.calcRefinancingAuctionRate()` returns an interest rate limit that increases as the auction progresses and has an upper limit. However, when `currentAuctionBlock` is greater than `auctionT2`, the interest rate limit can be over `_LIQUIDATION_THRESHOLD`.

In the annotated formula, the rate limit value is calculated with the below formula when `auctionT1 <= currentAuctionBlock < auctionT2`.

`g(x) = middleSlope * x + middleB = middleSlope * x + (maxRateWads - 6 * middleSlope) = maxRateWads + (x - 6) * middleSlope` where `middleB` is `maxRateWads - 6 * middleSlope` and `x` is the `(currentAuctionBlock * 30) / auctionDuration`

However, in the solidity implementation, the `g(x)` is calculated such as `g'(x) = maxRateWads * currentAuctionBlock + middleB = maxRateWads + (currentAuctionBlock - auctionT1) * middleSlope` where `middleB` is the `maxRateWads - auctionT1 * middleSlope` and `auctionT1` is the `auctionDuration / 5`.

Since `x - 6` and `currentAuctionBlock - auctionT1` are not equal, the calculation result will also differ. `x` has a value from 0 to 30, but the `currentAuctionBlock` can be up to 432000 (`_MAX_AUCTION_DURATION`).

In conclusion, the maximum rate limit is derived when `currentAuctionBlock` is `_MAX_AUCTION_DURATION` * 0.8, four times higher than `_LIQUIDATION_THRESHOLD`. For example, the interest rate limit is 4344% when the `currentAuctionBlock` is 345600, `auctionDuration` is 432000, and `oldRate` is 10%.

Impact

High

1. A borrower may get much more `debt` than expected. In the worst scenario, the maximum interest rate is 4344%. If the lien is refinanced at such a high-interest rate, interest more than the principal will accrue within just one week.
2. The rate limit can be much larger if `auctionDuration` exceeds `_MAX_AUCTION_DURATION`. An attacker can bypass the limit of `auctionDuration` using the `BLEND-003` vulnerability.

Recommendation

1. Modify the calculation for `middleB`
2. Change the `middleSlope` value. (It does not match the https://www.desmos.com/calculator/urasr71dhb)

References

- https://www.desmos.com/calculator/urasr71dhb
- Interest rate model graph `oldrate` : 100000, `y` : `rateLimit`, `x` : `auctionDuration`
- BLEND-003

Patch

Fixed

`middleSlope` calculation is corrected to scale for `auctionDuration`. As a result, `middleB` is also calculated accurately.

#9 `BLEND-009` Use two-step ownership transfer

ID	Summary	Severity
`BLEND-009`	When changing the owner, it is preferable to have the new owner send a transaction to accept the ownership, just in case the original owner enters the wrong address.	Informational

Description

The owner can change the owner with the `transferOwnership` function. However, if the owner is changed incorrectly by entering the wrong address, it cannot be taken back. Also, the contract will become un-upgradable because only the owner can perform the upgrade, so it is recommended to add a fail-safe.

Impact

Informational

Recommendation

Use `OZ.Ownable2StepUpgradeable` instead of `OZ.OwnableUpgradeable`.

References

https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol

Patch

Fixed

It is fixed as recommended.

Revision History

Version	Date	Description
1.0	April 27, 2023	Initial version of report
1.1	April 28, 2023	Fixed typo in BLEND-005

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.

ChainLight