# Arcade V4 Audit Report

Version 2.0


Audited by:

**MiloTruck**

**bytes032**

**alexxander**


February 22, 2024

# Contents

# 1 Introduction

## 1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

Check our portfolio here.

## 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

## 1.3 Risk Classification

| | **Impact: High** | **Impact: Medium** | **Impact: Low** |
|---|---|---|---|
| **Likelihood: High** | High | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

### 1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

### 1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

# 2 Executive Summary

## 2.1 About Arcade V4

Arcade.xyz is the first of its kind Web3 platform to enable liquid lending markets for NFTs. At Arcade.xyz, the belief is that all assets will eventually become digitized and that NFTs represent a 0 to 1 innovation in storing value and ownership attribution for unique digital assets.

Arcade.xyz's focus is on building primitives, infrastructure, and applications enabling the growth of NFTs as an asset class. As such, the first product they released is an innovative peer to peer lending marketplace that allows NFT owners to unlock liquidity on baskets of NFTs on Ethereum. Lenders that hold stablecoins or ERC20 tokens can participate in a new source of DeFi yield by underwriting term loans collateralized by borrowers' NFTs.

Arcade.xyz is an end user application that strives to become the premier liquidity venue for NFTs, via a protocol for NFT collateralized loans with flexible terms. Today NFTs are largely digital representations of artwork and media content, however, the team believes that in the not so distant future NFTs will encompass digital rights, metaverse assets, and digital identity. For more information about Arcade.xyz, please visit: `https://docs.arcadedao.xyz/docs`.

## 2.2 Overview

| | |
|---|---|
| Project | Arcade V4 |
| Repository | arcade-protocol |
| Commit Hash | 6901b412b208... |
| Mitigation Hash | 98cd3ad3b292... |
| Date | February 2024 |

## 2.3 Issues Found

| **Severity** | **Count** |
|---|---|
| High Risk | 7 |
| Medium Risk | 14 |
| Low Risk | 5 |
| Informational | 13 |
| **Total Issues** | **39** |

# 3 Findings Summary

| ID | Description | Status |
|---|---|---|
| H-1 | Reentrancy during migration allows for unauthorized access in `receiveFlashLoan()` | Resolved |
| H-2 | `RefinanceController._validateRefinance()` allows the new principal to be greater than the active balance of the old loan. | Resolved |
| H-3 | Borrower can abuse partial repayments to deny Lender from redeeming repaid funds. | Resolved |
| H-4 | Underflow in `InterestCalculator.sol` allows Borrower to default a loan and deny Lender from claiming collateral | Resolved |
| H-5 | Incorrect fee calculation when migrating V3 loans in `OriginationControllerMigrate.migrateV3Loan()` | Resolved |
| H-6 | `_validateCounterparties()` incorrectly checks if `sig` belongs to the `callingCounterparty` | Resolved |
| H-7 | `OriginationCalculator._calculateRolloverAmounts()` doesn't account for partially repaid loans | Resolved |
| M-01 | Permanent DOS of `OriginationControllerMigrate.migrateV3Loan()` after the first migration | Resolved |
| M-02 | In some cases a signature's nonce `maxUses` can be violated. | Resolved |
| M-03 | Borrower can force Lender to accrue `LENDER_REDEEM_FEE` | Resolved |
| M-04 | Missing principal fee charge in `OriginationCalculator._calculateRolloverAmounts()` | Resolved |
| M-05 | `LoanCore.rollver()` doesn't distribute fees to the old loan's affiliate | Resolved |
| M-06 | Borrowers can use `RepaymentController.forceRepay()` to temporarily prevent lenders from claiming collateral | Resolved |
| M-07 | Missing `signingCounterparty` in `loanHash` can lead to the signature being used with the wrong address | Resolved |
| M-08 | New lenders are unfairly charged for `interestFee` when rolling over an loan | Resolved |
| M-09 | The borrower's `callbackData` is not included in signatures | Resolved |
| M-10 | `migrateV3Loan()` will revert for lenders that are approved or use ERC-1271 | Resolved |
| M-11 | Stale `feeSnapshot` is used when rolling over a loan | Acknowledged |
| M-12 | `totalOwed` does not account for partial repayments | Resolved |
| M-13 | `lenderFee` and `interestFee` may not be collected from new lenders | Resolved |

| ID | Description | Status |
|---|---|---|
| M-14 | Reusable signatures can be repeatedly rolled over with `rolloverLoan()` to incur extra fees | Resolved |
| L-1 | Missing `isAllowedCollateral()` inside `OriginationControllerMigrate_validateV3Migration()` | Resolved |
| L-2 | `RefinanceController._validateRefinance()` missing `MIN_LOAN_DURATION` check | Resolved |
| L-3 | `RefinanceController._validateRefinance()` missing minimal principal check | Resolved |
| L-4 | Lender notes should not be traded on the secondary market | Resolved |
| L-5 | Lack of state update in `LoanCore.rollover()` can lead to inflated effective APR | Resolved |
| I-01 | Use a `MAX_LENGTH` constant for array length checks in `OriginationController.sol` | Resolved |
| I-02 | Errors in comments | Resolved |
| I-03 | Overriding `_unpause()` in `LoanCore.sol` is unnecessary | Resolved |
| I-04 | `aprMinimumScaled` calculation in `_validateRefinance()` can be simplified | Resolved |
| I-05 | Checking `data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE` in `_prepareRepay()` is redundant | Resolved |
| I-06 | `paymentToPrincipal` calculation can be unchecked in `_prepayRepay()` | Resolved |
| I-07 | Design improvement in fee query | Resolved |
| I-08 | Inconsistency in `RepaymentController.claim()` | Resolved |
| I-09 | `OriginationLibrary.isApprovedForContract()` should decode `result` as `bytes32` to avoid unexpected reverts in future use. | Resolved |
| I-10 | `setAffiliateSplits()` lacks `whenNotPaused` and `nonReentrant` modifiers | Resolved |
| I-11 | `amounts.amountFromLender > 0` check in `OriginationControllerMigrate._migrate()` is incorrect | Resolved |
| I-12 | Interest calculation in `getProratedInterestAmount()` can round down to 0 for tokens with low decimals | Resolved |
| I-13 | Stale data after claiming a loan | Resolved |

# 4 Findings

## 4.1 High Risk

### [H-1] Reentrancy during migration allows for unauthorized access in `receiveFlashLoan()`

**Context:**

- OriginationControllerMigrate.sol
- LoanCore.sol

**Description:** `OriginationControllerMigrate.migrateV3Loan()` has a `whenBorrowerReset` modifier that resets the `borrower` cache after the migration function completes. This is a measure to prevent external actors from initiating a flash loan with malicious data and targeting `OriginationControllerMigrate.receiveFlashLoan()`.

```
    function receiveFlashLoan(
        IERC20[] calldata assets,
        uint256[] calldata amounts,
        uint256[] calldata feeAmounts,
        bytes calldata params
    ) external nonReentrant {
        if (msg.sender != VAULT) revert OCM_UnknownCaller(msg.sender, VAULT);

        OriginationLibrary.OperationData memory opData = abi.decode(params,
        (OriginationLibrary.OperationData));

        // verify this contract started the flash loan
        if (opData.borrower != borrower) revert OCM_UnknownBorrower(opData.borrower,
        borrower);
        // borrower must be set
>       if (borrower == address(0)) revert OCM_BorrowerNotCached();

        _executeOperation(assets, amounts, feeAmounts, opData);
    }
```

The problem in `OriginationControllerMigrate.migrateV3Loan()` is that after `OriginationControllerMigrate._initiateFlashLoan()` finishes execution, the flash loan cycle that starts from `Balancer.flashLoan()` has already concluded and the reentrancy guards are unlocked.

However, execution continues in `OriginationControllerMigrate._initializeMigrationLoan()`, which will make a call to `LoanCore.startLoan()`, which then safe mints `ERC721 PromissoryNotes` to the lender and borrower for the new (migrated) loan. At this point, the borrower or lender can utilize the `onERC721Received` hook external call, during which they can initiate new flash loans through `Balancer.flashLoan()` with arbitrary data that targets `OriginationControllerMigrate.receiveFlashLoan()`, which won't revert since the `borrower` cache hasn't yet been reset by the `whenBorrowerReset` modifier.

The issue now is that `OriginationControllerMigrate._executeOperation()` can be called with arbitrary data, which can lead to impacts such as theft of funds from anyone who has approval towards `OriginationControllerMigrate` and abusing `safeApprove()` to permanently lock `OriginationControllerMigrate.migrateV3Loan()`.

```
function migrateV3Loan(
    ...
) external override whenNotPaused whenBorrowerReset {

    // code ...

    // @note borrower cache gets set in _initiateFlashLoan()
    // @audit whole flashloan cycle is contained within _initiateFlashLoan()
    if (flashLoanTrigger) {
        _initiateFlashLoan(oldLoanId, newTerms, msg.sender, lender, amounts);
    }

    // code ...

    // @audit At this point reentancy guards in this contract and in Balancer Vault
    are released
    // @audit borrower cache is not reset
    _initializeMigrationLoan(newTerms, msg.sender, lender, amounts.amountFromLender,
    amounts.amountToBorrower);

    // code ...
}
```

**Recommendation:** An alternative fix to the one below is to add `nonReentrant` modifier to `OriginationControllerMigrate. _initiateFlashLoan()`, however, the `borrower` cache would still remain initialized during `OriginationControllerMigrate.migrateV3Loan()`, even after the flash loan has been executed.

```
@@ -322,10 +322,12 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
             )
         );

         // Flash loan based on principal + interest
         IVault(VAULT).flashLoan(this, assets, amounts, params);
+
+        borrower = address(0);
     }

     /**
      * @notice Callback function for flash loan. OpData is decoded and used to
      execute the migration.
      *
@@ -481,12 +483,10 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
      */
     modifier whenBorrowerReset() {
         if (borrower != address(0)) revert OCM_BorrowerNotReset(borrower);

         _;
-
-        borrower = address(0);
     }
```

**Arcade**: Fixed in PR-92.

**Renascence:** The issue has been fixed as recommended.

**[H-2]** `RefinanceController._validateRefinance()` **allows the new principal to be greater than the active balance of the old loan.**

**Context:** RefinanceController.sol

**Description:** `RefinanceController._validateRefinance()` does the following check for the new loan's principal:

```
// principal cannot increase
if (newTerms.principal > oldLoanData.terms.principal) revert REFI_PrincipalIncrease(
    oldLoanData.terms.principal,
    newTerms.principal
);
```

This check, however, does not account for any previous partial repayments made towards the old loan. Therefore, it allows for refinancing loans where the new loan will have a higher principal than the active balance of the old loan.

The problem is that when the new loan's principal amount is more than the old loan's unrepaid amount (i.e. `newPrincipalAmount > oldLoanData.balance`), the borrower doesn't receive his repayment back.

For example:

- Assume that an old loan's principal is 100 USDC, and the borrower has repaid 50 USDC so far.
- For simple calcuations, we assume that all fees are 0%, and that the old loan has no interest to repay.
- If we rollover into a new loan with principal as 90 USDC, the numbers in `rolloverAmounts()` will be:

```
oldLoanPrincipal = 50
oldInterestAmount = 0
newPrincipalAmount = 90

borrowerOwedForNewLoan = newPrincipalAmount = 90
amountFromLender = newPrincipalAmount = 90

repayAmount = oldPrincipal + oldInterestAmount = 50
```

- Since `repayAmount > borrowerOwedForNewLoan` , the logic takes the `else` branch that repays funds to the borrower:

```
amountToBorrower = borrowerOwedForNewLoan - repayAmount = 40
```

After the calculations in `rolloverAmounts()`, the result is `amountToBorrower = 40`, which means the borrower should receive 40 USDC from `refinanceLoan()`. However, `LoanCore.rollover()` is called with `amountToBorrower` specified as `0`:

```
amounts.amountToOldLender,
0,
0, // amountToBorrower
```

So the borrower doesn't get his 40 USDC back. The accounting for the borrower is:

- He received 100 USDC for the old loan's principal.
- He repaid 50 USDC.
- After the loan is refinanced, he owes 90 USDC to the new lender.

Therefore, he is at a net loss of `100 - 50 - 90 = 40` USDC.

**Recommendation:** Restrict the new loan's principal to the old loan's unrepaid amount:

```
  // principal cannot increase
- if (newTerms.principal > oldLoanData.terms.principal) revert REFI_PrincipalIncrease(
+ if (newTerms.principal > oldLoanData.balance) revert REFI_PrincipalIncrease(
      oldLoanData.terms.principal,
      newTerms.principal
  );
```

**Arcade:** Fixed in PR-93

**Renascence:** The issue has been fixed as recommended.

#### [H-3] Borrower can abuse partial repayments to deny Lender from redeeming repaid funds.

**Context:** LoanCore.sol, RepaymentController.sol

**Description:** Borrowers can call `forceRepay()` instead of `repay()`, this will not transfer the repaid amount to the lender directly but is held in a `noteReceipts` mapping. Lenders can then call `redeemNote()` to claim the repaid amount, however, they still need to own their `PromissoryNote` to prove ownership.

```
function redeemNote(
    uint256 loanId,
    uint256 _amountFromLender,
    address to
) external override onlyRole(REPAYER_ROLE) nonReentrant {
    // Get owner of the LenderNote
    address lender = lenderNote.ownerOf(loanId);

    // if the loan has been completely repaid and no more repayments are expected
    if (loans[loanId].state == LoanLibrary.LoanState.Repaid) {
        // delete the receipt
        delete noteReceipts[loanId];

        // Burn ONLY the LenderNote
        lenderNote.burn(loanId);
    } else {
        // zero out the total amount owed in the receipt
        noteReceipts[loanId].amount = 0;
    }
}
```

However, since `repay()` doesn't check if the loan has an existing note receipt, a borrower can abuse partial repayment and note receipts to make the lender unable to claim the repaid principal + interest:

- Borrower calls `forceRepay()` to repay `principal + interest - 1 wei`. The repaid amount is held in a note receipt.
- Borrower calls `repay()` to repay the last 1 wei. This will burn the lender note since `loans[loanId].state` changes to `LoanState.Repaid`
- Lender can't call `redeemNote()` anymore since they no longer hold the lender note, so they can't ever claim the repayment.

Borrwers can also call `rollover()` in place of `repay()` to achieve the same impact, since `rollover()` also burns the lender note.

```
function repay(
    uint256 loanId,
    address payer,
    uint256 _amountToLender,
    uint256 _interestAmount,
    uint256 _paymentToPrincipal
) external override onlyRole(REPAYER_ROLE) nonReentrant {

    if (loans[loanId].state == LoanLibrary.LoanState.Repaid) {
        // if loan is completely repaid
        // burn both notes
        _burnLoanNotes(loanId);
        // redistribute collateral and emit event
        IERC721(data.terms.collateralAddress).safeTransferFrom(address(this),
        borrower, data.terms.collateralId);

        emit LoanRepaid(loanId);
    }
}
```

**Recommendation:** A potential would be to only burn the lender note when the loan's note receipt has no more balance:

```
@@ -923,7 +922,10 @@ contract LoanCore is
     function _burnLoanNotes(uint256 loanId) internal {
-        lenderNote.burn(loanId);
+        if (noteReceipts[loanId].amount == 0) {
+            lenderNote.burn(loanId);
+        }
+
         borrowerNote.burn(loanId);
     }
```

This way all functionality remains the same, except that if the lender hasn't called `redeemNote()` and a full repayment is made using `repay()` or `rollover()`, he will still have the ability to call `redeemNote()` afterwards.

Note that if the fix for - "Borrower can abuse *`RepaymentController.forceRepay()`* to temporary DOS a Lender from claiming collateral" - is to remove the check, you would have to burn lender notes in `redeemNote()` after `claim()` as well:

LoanCore.sol#L404-L405

```
  // if the loan has been completely repaid and no more repayments are expected
- if (loans[loanId].state == LoanLibrary.LoanState.Repaid) {
+ LoanLibrary.LoanState state = loans[loanId].state;
+ if (state == LoanLibrary.LoanState.Repaid || state ==
LoanLibrary.LoanState.Defaulted) {
```

**Arcade:** Fixed in PR-94

**Renascence:** The issue has been fixed as recommended.

### [H-4] Underflow in `InterestCalculator.sol` allows Borrower to default a loan and deny Lender from claiming collateral

**Context:** InterestCalculator.sol, RepaymentController.sol

**Description:** When a loan is past `loanStartTime + loanDuration`, a Lender can now call `RepaymentController.claim()` to claim the loan's collateral, but only after `loanStartTime + loanDuration + Constants.GRACE_PERIOD + 1` has passed.

The Borrower who has defaulted on the loan can now call `RepaymentController.repay()` and repay a minimum (the accrued interest), and `loans[loanId].lastAccrualTimestamp = uint64(block.timestamp)` will get updated.

This turns out to be a vulnerability because, after the Grace Period has passed, the Lender will try to call `RepaymentController.claim()`.

However, the call to `getProratedInterestAmount()` will try to fetch the interest on which a default fee is owed, but the code inside `InterestCalculator.getProratedInterestAmount()` will revert because `timeSinceStart > loanDuration` and `lastAccrualTimestamp > endTimestamp`.

Therefore InterestCalculator.sol#51 underflows, causing a revert, which means the Lender cannot claim the collateral indefinitely.

```
function getProratedInterestAmount(
    uint256 balance,
    uint256 interestRate,
    uint256 loanDuration,
    uint256 loanStartTime,
    uint256 lastAccrualTimestamp,
    uint256 currentTimestamp
) public pure returns (uint256 interestAmountDue) {
    // time since loan start
    uint256 timeSinceStart = currentTimestamp - loanStartTime;

    // time since last payment
    uint256 timeSinceLastPayment;
    if (timeSinceStart > loanDuration) {
        // if time elapsed is greater than loan duration, set it to loan duration
        uint256 endTimestamp = loanStartTime + loanDuration;

>       timeSinceLastPayment = endTimestamp - lastAccrualTimestamp;
    } else {
        timeSinceLastPayment = currentTimestamp - lastAccrualTimestamp;
    }
    interestAmountDue = balance * timeSinceLastPayment * interestRate
        / (Constants.BASIS_POINTS_DENOMINATOR * Constants.SECONDS_IN_YEAR);
}
```

**Note:** Even with a `GRACE_PERIOD = 0` this is still an issue because the lender can only claim when:

```
uint256 dueDate = data.startDate + data.terms.durationSecs + Constants.GRACE_PERIOD;
if (dueDate >= block.timestamp) revert LC_NotExpired(dueDate);
```

This means the timestamp of claiming is always > `endTimestamp` (calculated in `InterestCalculator`).

As a resultthe issue still exists, however, with the added complexity of the Borrower front-running the Lender with a `repay()`.

**Recommendation**

In `getProratedInterestAmount()`, when `lastAccrualTimestamp` is greater than the loan's `endTimestamp`, return the amount of interest owed as `0`:

```
@@ -47,10 +47,14 @@ abstract contract InterestCalculator {
         uint256 timeSinceLastPayment;
         if (timeSinceStart > loanDuration) {
             // if time elapsed is greater than loan duration, set it to loan duration
             uint256 endTimestamp = loanStartTime + loanDuration;

+            if(lastAccrualTimestamp >= endTimestamp) {
+                return 0;
+            }
+
             timeSinceLastPayment = endTimestamp - lastAccrualTimestamp;
         } else {
             timeSinceLastPayment = currentTimestamp - lastAccrualTimestamp;
         }
```

**Arcade:** Fixed in PR-95.

**Renascence:** The issue has been fixed as recommended.

### [H-5] Incorrect fee calculation when migrating V3 loans in `OriginationControllerMigrate.migrateV3Loan()`

**Context:**

- OriginationControllerMigrate.sol#L452-L453
- OriginationControllerMigrate.sol#L121-L122

**Description:** `OriginationControllerMigrate.migrateV3Loan()` calls `_initializeMigrationLoan()` with `amounts.amountFromLender` and `amounts.amountToBorrower`:

```
// initialize v4 loan
_initializeMigrationLoan(newTerms, msg.sender, lender, amounts.amountFromLender,
amounts.amountToBorrower);
```

However, `amounts.amountFromLender` and `amounts.amountToBorrower` here are different from the expected values of `_amountFromLender` and `_amountToBorrower` in `LoanCore.startLoan()`.

`_initializeMigrationLoan()` directly passes `amounts.amountsFromLender` and `amounts.amountToBorrower` into `startLoan()`, which calculates the fees earned by the protocol using the difference of both values:

```
// Assign fees for withdrawal
uint256 feesEarned;
unchecked { feesEarned = _amountFromLender - _amountToBorrower; }
```

`amountFromLender - amountToBorrower` here is actually not equal to the fees earned from migrating a V3 loan.

According to `rolloverAmounts()`:

```
amountFromLender = newPrincipalAmount + lenderFee + interestFee
```

When `repayAmount > borrowerOwedForNewLoan`:

```
amountToBorrower = 0
amountFromLender - amountToBorrower = newPrincipalAmount + lenderFee + interestFee
```

Otherwise, with some simple algebra:

```
amountToBorrower = borrowerOwedForNewLoan - repayAmount
amountFromLender - amountToBorrower = repayAmount + lenderFee + interestFee +
borrowerFee
```

As seen from above, `amountFromLender - amountToBorrower` will always include `newPrincipalAmount` or `repayAmount`. Additionally, `borrowerFee` will be excluded when `repayAmount > borrowerOwedForNewLoan`.

This will cause `feesEarned` to end up becoming a largely inflated value. Since fees are distributed between the protocol and affiliates, affiliates will be able to withdraw more fees than intended, causing a loss of fees for the protocol.

**Recommendation:** When calling `startLoan()` in `_initializeMigrationLoan()`, pass `amountFromLender` as `borrowerFee + lenderFee` and `amountToBorrower` as `0`:

```
  // create loan in LoanCore
- newLoanId = loanCore.startLoan(lender, borrower_, newTerms, amountFromLender,
amountToBorrower, feeSnapshot);
+ newLoanId = loanCore.startLoan(lender, borrower_, newTerms, borrowerFee + lenderFee,
0, feeSnapshot);
```

Consider removing the `amountFromLender` and `amountToBorrower` parameters from `_initializeMigrationLoan()` as they are no longer needed:

```
  function _initializeMigrationLoan(
      LoanLibrary.LoanTerms memory newTerms,
      address borrower_,
-     address lender,
+     address lender
-     uint256 amountFromLender,
-     uint256 amountToBorrower
  ) internal returns (uint256 newLoanId) {
```

In `migrateV3Loan()`:

```
  // initialize v4 loan
- _initializeMigrationLoan(newTerms, msg.sender, lender, amounts.amountFromLender,
amounts.amountToBorrower);
+ _initializeMigrationLoan(newTerms, msg.sender, lender);
```

**Arcade:** Fixed in PR-96. A refactoring is introduced where `LoanCore.startLoan()` now takes a `feesEarned` param, calculated in the OC contracts.

**Renascence-Labs:** The issue has been fixed as recommended.

### [H-6] `_validateCounterparties()` incorrectly checks if `sig` belongs to the `callingCounterparty`

**Context:** OriginationController.sol#L436-L438

**Description:** `_validateCounterparties()` validates that the caller is allowed to create a new loan for the `lender` and `borrower` addresses. This is done by checking that the caller is authorized create loans on the behalf of one party, known as the `callingCounterparty`, and the signature provided belongs to the other party, known as the `signingCounterparty`.

However, `_validateCounterparties()` incorrectly checks if the signature belongs to the `callingCounterparty` as such:

```
if (!isSelfOrApproved(callingCounterparty, caller) &&
!OriginationLibrary.isApprovedForContract(callingCounterparty, sig, sighash)) {
    revert OC_CallerNotParticipant(msg.sender);
}
```

The `!OriginationLibrary.isApprovedForContract(callingCounterparty, sig, sighash)` condition should not be there as the signature should belong to only the `signingCounterparty`, and **not** the `callingCounterparty`.

An attacker can abuse this to force a user with a `Side.BORROW` signature to become a lender:

- Assume Bob uses a smart contract wallet:
  - Bob's wallet has an existing approval of 1000 USDC to the `LoanCore` contract.
- Bob wants to borrow a loan using his Azuki NFT as collateral. He generates a signature with the following:
  - `side = Side.BORROW`

  - `loanTerms.collateralAddress` is the Azuki address
  - `loanTerms.collateralId = 1337`
  - `loanTerms.payableCurrency` is the USDC address
  - `loanTerms.principal = 1000e6`. Note that this amount can be anything smaller than Bob's approval.
- Alice wishes to make Bob become a lender for the loan, she does the following:
  - Deploy a malicious contract that will be the `borrower` address. Whenever `isValidSignature()` is called, its selector is always returned.
  - She buys Bob's Azuki NFT from him and transfers it to the malicious contract.
  - Before Bob can invalidate his signature using `cancelNonce()` in `LoanCore`, she calls `initializeLoan()` with the following arguments:
    * `loanTerms` is set to the terms in Bob's signature.
    * `borrower` is Alice's malicious contract.
    * `lender` is Bob's wallet.
    * `sig` is set to Bob's signature.
    * `nonce` is set to the nonce in Bob's signature.
  - Note that the caller (`msg.sender`) is Alice's address.
- In `initializeLoan()`:
  - `neededSide = Side.BORROW`, as Alice's malicious contract is not approved by her address.
  - In `_validateCounterparties()`:
    * `signingCounterparty` is Alice's malicious contract.
    * `callingCounterparty` is Bob's wallet.
    * The `isApprovedForContract()` check for `callingCounterparty` passes, as Bob is the signer of the BORROW signature.
    * The `isApprovedForContract()` check for `signingCounterparty` passes, as Alice's malicious contract returns the correct magic value for any signature.
- As a result, a new loan is created with Bob's wallet as the lender.

In the scenario above, Bob's signature was used to force him to become a lender, even though his signature contained `Side.BORROW`.

Note that the likelihood of a user having an existing approval to the `LoanCore` contract is not low; the user could be waiting for another loan to be created, where he is the lender that provides currency.

The exploit shown above can be used to manipulate signatures using any function that calls `_validateCounterparties()`, including:

- `initializeLoan()`
- `initializeLoanWithItems()`
- `rolloverLoan()`
- `rolloverLoanWithItems()`

**Recommendation:** Remove the `isApprovedForContract()` condition for `callingCounterparty`:

```
-       if (!isSelfOrApproved(callingCounterparty, caller) &&
!isApprovedForContract(callingCounterparty, sig, sighash)) {
+       if (!isSelfOrApproved(callingCounterparty, caller)) {
            revert OC_CallerNotParticipant(msg.sender);
        }
```

**Arcade:** Fixed in PR-97.

**Renascence:** The issue has been fixed as recommended.

### [H-7] `OriginationCalculator._calculateRolloverAmounts()` doesnt account for partially repaid loans

**Context:** OriginationCalculator.sol#L141-L143

**Description:** `OriginationCalculator._calculateRolloverAmounts()` calls `rolloverAmounts()` with `oldLoanData.terms.principal`, which is the loan's entire principal:

```
return rolloverAmounts(
    oldLoanData.terms.principal,
    interest,
```

However, it should call `rolloverAmounts()` with `oldLoanData.balance` instead as partial repayments are now possible in V4. The borrower could have repaid part of the old loan's principal before it is rolled over into a new loan.

For rollovers, this will cause the borrower to overpay when `rolloverLoan()` is called since `repayAmount`, the amount of principal + interest the borrower needs to repay for the old loan, will be higher than it should be. This is highly likely to occur when `borrowerOwedForNewLoan > repayAmount` in `rolloverAmounts()` as `rolloverLoan()` will not transfer any funds from the borrower.

For refinancing loans, `RefinanceController.refinanceLoan()` will incorrectly pull the whole `oldLoanData.terms.principal` from the new lender and overpay to the old lender.

For example:

- Borrower is lent `1e18`, i.e. `terms.principal` is `1e18` (assuming 0 fees or interest)
- Borrower repays `5e17`, i.e borrower has `5e17` and lender has `5e17`
- A new lender decided to refinance with smaller interest but the same principal of `1e18`, i.e. has to repay `5e17` to the old lender and transfer `5e17` to the borrower
- `_calculateRolloverAmounts()` calculates `amountToOldLender = 1e18` - the old lender is overpaid (now has `1e18 + 5e17`) and borrower doesn't receive anything, leaving him with `5e17`

**Recommendation:** Replace `oldLoanData.terms.principal` with `oldLoanData.balance`, which is the amount of principal yet to be repaid:

```
  return rolloverAmounts(
-     oldLoanData.terms.principal,
+     oldLoanData.balance,
```

**Arcade:** Fixed in PR-98

**Renascence:** The issue has been fixed as recommended.

## 4.2 Medium Risk

**[M-01] Permanent DOS of `OriginationControllerMigrate.migrateV3Loan()` after the first migration**

**Context:** OriginationControllerMigrate.sol, V3RepaymentController

**Description:** During `OriginationControllerMigrate.migrateV3Loan()`, a call is made to `OriginationControllerMIgrate.rolloverAmounts()` to calculate the `OriginationLibrary.RolloverAmounts` needed for the migration.

`OriginationControllerMIgrate.rolloverAmounts()` includes the `lenderFee` and `borrowerFee` inside `amounts.amountFromLender` and `amounts.needFromBorrower`.

This becomes an issue when later the `repayAmount` passed as a parameter to `OriginationControllerMigrate_repayLoan()` is calculated with the values that include the fees.

```
_repayLoan(msg.sender, IERC20(newTerms.payableCurrency), oldLoanId,
amounts.amountFromLender + amounts.needFromBorrower - amounts.amountToBorrower);
```

Later in `OriginationControllerMigrate_repayLoan()` we have the line:

```
// approve LoanCoreV3 to take the total settled amount
payableCurrency.safeApprove(loanCoreV3, repayAmount);

// repay V3 loan, this contract receives the collateral
IRepaymentControllerV3(repaymentControllerV3).repay(borrowerNoteId);
```

The repayment mechanism in V3 will pull from `OriginationControllerMigrate` only the owed `principal + interest`, therefore, because the approved amount included the borrower and lender "origination" fees, `OriginationControllerMigrate` will have a `> 0` approval left towards V3.

This will make migrations for the particular `payableCurrency` impossible since on the next attempt to migrate, `payableCurrency.safeApprove(loanCoreV3, repayAmount)` will revert with `SafeERC20: approve from non-zero to non-zero allowance`.

For example:

```
Borrower owes $100 + $20 interest to Lender 1 in V3
Assume $5 per origination fee in V4
Borrwer migrates with Lender 2 for the same $100 principal

borrowerOwedForNewLoan = $100 - $5 = $95
amounts.amountFromLender = $100 + $5 = $105
repayAmount = $100 + $20 = $120
amounts.needFromBorrower = $120 - $95 = $25

in _repayLoan(),
repayAmount = amounts.amountFromLender + amounts.needFromBorrower
repayAmount = $105 + $25 = $130

_repayLoan() approves $130 to V3
V3 pulls $120 (principal + interest)
excess approval = $10

Future calls to migrateV3Loan() are not possible since safeApprove()
reverts with "SafeERC20: approve from non-zero to non-zero allowance"
```

**Recommendation:** The suggested fix is to calculate the repayment amount for the V3 loan using the `oldTerms` when the old loan's interest amount is fetched in `OriginationControllerMigrate._calculateV3MigrationAmounts()`.

Then, the calculated repay amount will be returned upwards to `OriginationControllerMigrate.migrateV3Loan()` and passed over to `OriginationControllerMigrate._initiateFlashLoan()` or `OriginationControllerMigrate._repayLoan()`.

```
@@ -103,14 +103,15 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
         // collect and distribute settled amounts
         (
             OriginationLibrary.RolloverAmounts memory amounts,
-            bool flashLoanTrigger
+            bool flashLoanTrigger,
+            uint256 repayAmount
         ) = _migrate(oldLoanId, oldLoanData, newTerms.principal, msg.sender, lender);

         // repay v3 loan
         if (flashLoanTrigger) {
-            _initiateFlashLoan(oldLoanId, newTerms, msg.sender, lender,  amounts);
+            _initiateFlashLoan(oldLoanId, newTerms, msg.sender, lender,  amounts,
repayAmount);
         } else {
-            _repayLoan(msg.sender, IERC20(newTerms.payableCurrency), oldLoanId,
amounts.amountFromLender + amounts.needFromBorrower - amounts.amountToBorrower);
+            _repayLoan(msg.sender, IERC20(newTerms.payableCurrency), oldLoanId,
repayAmount);

             if (amounts.amountToBorrower > 0) {
                 // If new principal is greater than old loan repayment amount, send
                 the difference to the borrower
```

```
@@ -204,7 +205,8 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
         address lender
     ) internal nonReentrant returns (
         OriginationLibrary.RolloverAmounts memory amounts,
-        bool flashLoanTrigger
+        bool flashLoanTrigger,
+        uint256 repayAmount
     ) {
         address oldLender = ILoanCoreV3(loanCoreV3).lenderNote().ownerOf(oldLoanId);
         IERC20 payableCurrency = IERC20(oldLoanData.terms.payableCurrency);
@@ -213,7 +215,7 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
         (, uint256 borrowerFee, uint256 lenderFee) =
         feeController.getOriginationFeeAmounts(newPrincipalAmount);

         // Calculate settle amounts
-        (amounts) = _calculateV3MigrationAmounts(
+        (amounts, repayAmount) = _calculateV3MigrationAmounts(
             oldLoanData,
             newPrincipalAmount,
             lender,
```

```
@@ -260,13 +262,16 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
         address oldLender,
         uint256 borrowerFee,
         uint256 lenderFee
-    ) internal view returns (OriginationLibrary.RolloverAmounts memory amounts) {
+    ) internal view returns (OriginationLibrary.RolloverAmounts memory, uint256
repayAmount) {
         // get total interest to close v3 loan
         uint256 interest =
         IRepaymentControllerV3(repaymentControllerV3).getInterestAmount(
             oldLoanData.terms.principal,
             oldLoanData.terms.proratedInterestRate
         );

+        // calculate the repay amount to settle V3 loan
+        repayAmount = oldLoanData.terms.principal + interest;
+
        return(
             rolloverAmounts(
                 oldLoanData.terms.principal,
```

```
@@ -277,7 +282,7 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
                 borrowerFee,
                 lenderFee,
                 0
-            )
+            ), repayAmount
         );
     }

@@ -298,7 +303,8 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
         LoanLibrary.LoanTerms memory newLoanTerms,
         address borrower_,
         address lender,
-        OriginationLibrary.RolloverAmounts memory _amounts
+        OriginationLibrary.RolloverAmounts memory _amounts,
+        uint256 repayAmount
     ) internal {
         // cache borrower address for flash loan callback
         borrower = borrower_;
```

```
@@ -308,7 +314,7 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,

         // flash loan amount = new principal + any difference supplied by borrower
         uint256[] memory amounts = new uint256[](1);
-        amounts[0] = _amounts.amountFromLender + _amounts.needFromBorrower -
_amounts.amountToBorrower;
+        amounts[0] = repayAmount;

         bytes memory params = abi.encode(
             OriginationLibrary.OperationData(
```

**Arcade:** Fixed in PR-105.

**Renascence:** The issue has been fixed as recommended.

#### [M-02] In some cases a signatures nonce `maxUses` can be violated.

**Context:** LoanCore.sol

**Description:** In a signature, the signer specifies a `maxUses` value representing how many times a signature can be used.

The `LoanCore._useNonce()` function will mark a nonce as "used" when it has been used `maxUses` times.

```
function _useNonce(address user, uint160 nonce, uint96 maxUses) internal {
    // load nonce data
    mapping(uint160 => bool) storage _usedNonces = usedNonces[user];
    uint96 _nonceUses = numNonceUses[user][nonce];

    // check if nonce has been completely used or cancelled
    if (_usedNonces[nonce]) revert LC_NonceUsed(user, nonce);

    if (_nonceUses + 1 == maxUses) {
        // if this is the last time nonce can be used, mark the nonce as completely
        used
        // and update the number of times it has been used to the maxUses
        _usedNonces[nonce] = true;
        numNonceUses[user][nonce] = maxUses;

        emit NonceUsed(user, nonce);
    } else {
        // if this nonce usage is not the last use and is not over the maxUses,
        // increment the numNonceUses mapping
        numNonceUses[user][nonce]++;
    }
}
```

However, if `maxUses <= _nonceUses`, the signature can be used infinitely as `_nonceUses + 1 == maxUses` will never be true.

This could occur if:

1. The user creates a signature with `maxUses = 0`
2. The user re-uses a nonce that was used in an old signature that didn't reach `maxUses`

Another example would be:

- The user creates a signature with `nonce = 1` and `maxUses = 5`
- The signature is used 4 times
- The user signs a new signature with `nonce = 1` and `maxUses = 2`
- Since `_nonceUses = 4` and `maxUses` will always be smaller than `_nonceUses + 1`, the second signature can be used infinitely many times.

**Recommendation:** Consider reverting when `maxUses` is not greater than `nonceUses`:

```
@@ -883,7 +883,7 @@ contract LoanCore is
         uint96 _nonceUses = numNonceUses[user][nonce];

         // check if nonce has been completely used or cancelled
-        if (_usedNonces[nonce]) revert LC_NonceUsed(user, nonce);
+        if (_usedNonces[nonce] || maxUses <= _nonceUses) revert LC_NonceUsed(user,
nonce);

         if (_nonceUses + 1 == maxUses) {
```

**Arcade:** Fixed in PR-106.

**Renascence:** The issue has been fixed as recommended.

#### [M-03] Borrower can force Lender to accrue `LENDER_REDEEM_FEE`

**Context:** RepaymentController.sol

**Description:** The `LENDER_REDEEM_FEE`, which is `FL_08` in `FeeController.sol`, is a fee incurred by the Lender whenever they use `RepaymentController.redeemNote()` to redeem funds that the Borrower didn't send to them directly.

The issue is that the Borrower can always choose to use `RepaymentController.forceRepay()` to force the Lender to pay the `FL_08` fee. This can be considered a form of griefing, since the Borrower can force the Lender to incur fees at no additional cost.

Additionally, this fee is not stored in the loan's `feeSnapshot` on loan creation. As such, if the protocol increases the redemption fee during the loan's lifetime, the updated fee value will be used instead of the fee that the Lender agreed to upon loan creation..

**Recommendation:** Consider removing the `LENDER_REDEEM_FEE`.

```
@@ -180,9 +180,7 @@ contract RepaymentController is IRepaymentController,
InterestCalculator, FeeLoo
         address lender = lenderNote.ownerOf(loanId);
         if (lender != msg.sender) revert RC_OnlyLender(lender, msg.sender);

-        uint256 redeemFee = (amountOwed * feeController.getLendingFee(FL_08)) /
Constants.BASIS_POINTS_DENOMINATOR;
-
-        loanCore.redeemNote(loanId, redeemFee, to);
+        loanCore.redeemNote(loanId, 0, to);
     }
```

Otherwise, include the redemption fee in `feeSnapshot`, which ensures that the redemption fee was agreed upon by the lender on loan creation.

**Arcade:** Fixed in PR-107.

**Renascence:** The issue has been fixed as recommended.

#### [M-04] Missing principal fee charge in `OriginationCalculator._calculateRolloverAmounts()`

**Context:** OriginationCalculator.sol, RepaymentController.sol

**Description:** The repayment functions `RepaymentController.repay()` and `RepaymentController.forceRepay()` charge the `feeSnapshot.lenderPrincipalFee` in favor of the affiliate and the protocol.

The issue is that `OriginationController._rollover` and `RefinanceController._refinance()` call `OriginationCalculator._calculateRolloverAmounts()` where `feeSnapshot.lenderPrincipalFee` is not charged.

```
function _calculateRolloverAmounts(
    LoanLibrary.LoanData memory oldLoanData,
    uint256 newPrincipalAmount,
    address lender,
    address oldLender,
    IFeeController feeController
) internal view returns (OriginationLibrary.RolloverAmounts memory) {
    //...

    // Calculate amount to be sent to borrower for new loan minus rollover fees
    uint256 borrowerFee = (newPrincipalAmount * feeData.borrowerRolloverFee) /
    Constants.BASIS_POINTS_DENOMINATOR;

    // Calculate amount to be collected from the lender for new loan plus rollover
    fees
    uint256 interestFee = (interest * oldLoanData.feeSnapshot.lenderInterestFee) /
    Constants.BASIS_POINTS_DENOMINATOR;
    uint256 lenderFee = (newPrincipalAmount * feeData.lenderRolloverFee) /
    Constants.BASIS_POINTS_DENOMINATOR;

    // The fee `oldLoanData.feeSnapshot.lenderPrincipalFee` is not charged
    return rolloverAmounts(
        // ...
    );
}
```

This causes a loss of fees for the protocol and the old/new loan's affiliate since they don't receive the principal fee from the final repayment.

Depending on the rollover fees, it could even be cheaper for borrowers to rollover their loan into a smaller one with themselves as the lender to avoid paying principal fees, rather than repaying with `repay()` or `forceRepay()`.

**Recommendation:** Add the `principalFee` and `interestFee` to a variable named `feeFromOldLender` and pass it in place of `interestFee`:

```
@@ -130,23 +130,25 @@ abstract contract OriginationCalculator is InterestCalculator {
             block.timestamp
         );

         // Calculate amount to be sent to borrower for new loan minus rollover fees
         uint256 borrowerFee = (newPrincipalAmount * feeData.borrowerRolloverFee) /
         Constants.BASIS_POINTS_DENOMINATOR;

         // Calculate amount to be collected from the lender for new loan plus
         rollover fees
         uint256 interestFee = (interest * oldLoanData.feeSnapshot.lenderInterestFee)
         / Constants.BASIS_POINTS_DENOMINATOR;
         uint256 lenderFee = (newPrincipalAmount * feeData.lenderRolloverFee) /
         Constants.BASIS_POINTS_DENOMINATOR;

+        uint256 principalFee = (oldLoanData.balance *
oldLoanData.feeSnapshot.lenderPrincipalFee) / Constants.BASIS_POINTS_DENOMINATOR;
+        uint256 feeFromOldLender = interestFee + principalFee;

         return rolloverAmounts(
             oldLoanData.terms.principal,
             interest,
             newPrincipalAmount,
             lender,
             oldLender,
             borrowerFee,
             lenderFee,
-            interestFee
+            feeFromOldLender
         );
     }
 }
```

**Arcade:** Instead of this fix, we chose to refactor the fees and how they are assessed. There are no more 'rollover fees' only origination fees everywhere a loan gets started, startLoan, rollovers, migrations, refinancing… . We are not going to address the principal fee issue here since it would essentially be doubling the fee on the new principal amount.

Code refactor in PR-108.

**Renascence:** Agree with the refactored code, the issue has been acknowledged.

#### [M-05] `LoanCore.rollver()` doesnt distribute fees to the old loans affiliate

**Context:** LoanCore.sol#L473-L475

**Description:** `LoanCore.rollover()` distributes fees to only the new loan's `affiliateCode` and doesn't use old loan's `affiliateCode`:

```
// Make sure split goes to affiliate code from _new_ terms
(uint256 protocolFee, uint256 affiliateFee, address affiliate) =
    _getAffiliateSplit(feesEarned, terms.affiliateCode);
```

Since the new loan's `affiliateCode` can be different from the old loan, the old loan's affiliate will

lose out on interest fees when the loan is rolled over. These fees will be distributed to the new affiliate instead.

**Recommendation:** A possible fix would be to distinguish between ”repayment fees” and ”rollover fees”. With reference to:

OriginationCalculator.sol#L133-L138

```
// Calculate amount to be sent to borrower for new loan minus rollover fees
uint256 borrowerFee = (newPrincipalAmount * feeData.borrowerRolloverFee) /
Constants.BASIS_POINTS_DENOMINATOR;

// Calculate amount to be collected from the lender for new loan plus rollover fees
uint256 interestFee = (interest * oldLoanData.feeSnapshot.lenderInterestFee) /
Constants.BASIS_POINTS_DENOMINATOR;
uint256 lenderFee = (newPrincipalAmount * feeData.lenderRolloverFee) /
Constants.BASIS_POINTS_DENOMINATOR;
```

`interestFee` and `principalFee` are repayment fees, while `borrowerFee` and `lenderFee` are rollover fees. In `LoanCore.rollover()`, instead of calculating fees using:

```
feesEarned = _settledAmount - _amountToOldLender - _amountToLender - _amountToBorrower
```

The protocol could add two parameters named `repaymentFee` and `rolloverFee`, which are passed from `OriginationController._rollover()`. `repaymentFee` would then be split between the old loan's affiliate and the protocol, while `rolloverFee` is split between the new loan's affiliate and the protocol.

Note that all rollover-related functions in `OriginationController.sol` will have to be refactored to accommodate this change as well.

**Arcade:** Addressed in PR-108.

**Renascence:** Agree with the refactored code, the issue has been fixed.

### [M-06] Borrowers can use `RepaymentController.forceRepay()` to temporarily prevent lenders from claiming collateral

**Context:** RepaymentController.sol, LoanCore.sol

**Description:** If a loan is past its due date, the Lender can claim the Borrower's collateral, which will end up with a call to `LoanCore.claim()`.

```
function claim(uint256 loanId, uint256 _amountFromLender)
    external
    override
    onlyRole(REPAYER_ROLE)
    nonReentrant
{
    LoanLibrary.LoanData memory data = loans[loanId];
    // Ensure valid initial loan state when claiming loan
    if (data.state != LoanLibrary.LoanState.Active) revert
    LC_InvalidState(data.state);

    // Check that loan has a noteReceipt of zero amount
    if (noteReceipts[loanId].amount != 0) revert
    LC_AwaitingWithdrawal(noteReceipts[loanId].amount);

    // ...
}
```

Due to the `noteReceipts[loanId].amount != 0` check above, the function will revert if the loan's note receipt has an outstanding balance that has not yet been claimed by the lender.

A borrower can take advantage of this check to force `claim()` to revert:

- Lender calls `claim()`.
- Borrower front-runs the lender's transaction and calls `forceRepay()` with 1 wei to create a note receipt.
- Lender's transaction is executed, but `claim()` reverts due to the check above. He has to call `redeemNote()` first before trying again.

As a result, the Borrower can easily prolong the loan for the cost of gas fees, although their loan duration is over.

Note that there's no DOS for a fixed duration, but the borrower is actually incentivised to abuse this to delay the lender from claiming his collateral. He can also repeat this attack for as long as he wants.

**Recommendation:**

Consider removing the `noteReceipts[loanId].amount != 0` check:

```
@@ -333,9 +333,6 @@ contract LoanCore is
         // Ensure valid initial loan state when claiming loan
         if (data.state != LoanLibrary.LoanState.Active) revert
         LC_InvalidState(data.state);

-        // Check that loan has a noteReceipt of zero amount
-        if (noteReceipts[loanId].amount != 0) revert
LC_AwaitingWithdrawal(noteReceipts[loanId].amount);

         // First check if the call is being made after the due date plus 10 min grace
         period.
```

**Arcade:** Fixed in PR-94.

**Renascence:** The issue has been fixed as recommended.

### [M-07] Missing `signingCounterparty` in `loanHash` can lead to the signature being used with the wrong address

**Context:** OriginationController.sol

**Description:** `loanHash` is used in recoverTokenSignature`and`recoverItemsSignature‘ to determine the external signer for a signature specifying only a collateral address and ID.

```
bytes32 loanHash = keccak256(
            abi.encode(
                OriginationLibrary._TOKEN_ID_TYPEHASH,
                loanTerms.interestRate,
                loanTerms.durationSecs,
                loanTerms.collateralAddress,
                loanTerms.deadline,
                loanTerms.payableCurrency,
                loanTerms.principal,
                loanTerms.collateralId,
                loanTerms.affiliateCode,
                sigProperties.nonce,
                sigProperties.maxUses,
                uint8(side)
            )
        );
```

However, it doesn't include the address of the `signingCounterparty`. Depending on the side, this could be the `lender` or the `borrower` address.

This makes it possible for the recently disclosed ERC1271 bug to occur, where if a signer's smart wallet's `isValidSignature()` follows the reference implementation in ERC-1271, their signature can be used with the wrong address as the `signingCounterparty`.

Consider the following example:

- Bob wants to be a lender with his smart wallet - he signs a signature with his EOA, which will return `true` when passed to his smart wallet's `isValidSignature()`.
- Alice calls `initializeLoan()` with his signature but with his EOA as the `lender` address.
- `isSelfOrApproved()` in `_validateCounterparties()` passes since`signer` is Bob's EOA.

An important note is that the signature can't be replayed in this case, so the only impact is that it can be used with the wrong address.

**Recommendation:** Include the `signingCounterparty` in the signature:

```
    function initializeLoan(
        LoanLibrary.LoanTerms calldata loanTerms,
        BorrowerData calldata borrowerData,
        address lender,
        Signature calldata sig,
        SigProperties calldata sigProperties,
        LoanLibrary.Predicate[] calldata itemPredicates
    ) public override returns (uint256 loanId) {
+         address signingCounterparty = neededSide == Side.LEND ? lender : borrower
+       (bytes32 sighash, address externalSigner) = _recoverSignature(loanTerms, sig,
sigProperties, signingCounterparty, neededSide, itemPredicates);
```

This stops the attack described above, as the `signer` and `sighash` would change if `signingCounterparty` were a different address. Hence, the signature check in `_validateCounterparties()` would fail.

**Arcade:** Fixed in PR-109.

**Renascence:** The issue has been fixed as recommended.

#### [M-08] New lenders are unfairly charged for `interestFee` when rolling over an loan

**Context:**

- OriginationCalculator.sol#L47-L53
- RepaymentController.sol#L244-L245

**Description:** In `rolloverAmounts()`, `interestFee` is added to `amountFromLender`:

```
if (borrowerFee > 0 || lenderFee > 0 || interestFee > 0) {
    unchecked {
        borrowerOwedForNewLoan = newPrincipalAmount - borrowerFee;
        amounts.amountFromLender = newPrincipalAmount + lenderFee + interestFee;
    }
} else {
```

However, this means that the new lender incurs the interest fee from the old loan, which is not ideal as the lender of the new loan should not be charged for anything related to the old loan.

As a result, the new lender will be forced to pay more when entering a loan using `rolloverLoan()` as compared to `initializeLoan()`.

For comparison, in `RepaymentController._prepareRepay()`, which is used in `repay()` and `forceRepay()`, the interest fee is charged to the loan's lender:

```
// the amount to send to the lender
amountToLender = amountFromBorrower - interestFee - principalFee;
```

**Recommendation:** In `rolloverAmounts()`, add `interestFee` to `amountFromLender` only when the old and new lender are the same:

```
  if (borrowerFee > 0 || lenderFee > 0 || interestFee > 0) {
      unchecked {
          borrowerOwedForNewLoan = newPrincipalAmount - borrowerFee;
-         amounts.amountFromLender = newPrincipalAmount + lenderFee + interestFee;
+         amounts.amountFromLender = newPrincipalAmount + lenderFee;
+
+         if (lender == oldLender) {
+             amounts.amountFromLender += interestFee;
+         }
      }
  } else {
```

In the scenario where the new and old lender are different, subtract `interestFee` from the amount repaid to the old lender:

OriginationCalculator.sol#L78-L81

```
  // Calculate lender amounts based on if the lender is the same as the old lender
  if (lender != oldLender) {
      // different lenders, repay old lender
-     amounts.amountToOldLender = repayAmount;
+     amounts.amountToOldLender = repayAmount - interestFee;
```

**Arcade:** Fixed in PR-110.

**Renascence:** The issue has been fixed as recommended.

#### [M-09] The borrowers `callbackData` is not included in signatures

**Context:** OriginationController

**Description:** When initializing a loan, the callee can assign `callbackData` to the `borrowerData` to execute an operation once the loan is initialized.

```
function initializeLoan(
       LoanLibrary.LoanTerms calldata loanTerms,
       BorrowerData calldata borrowerData,
       address lender,
       Signature calldata sig,
       SigProperties calldata sigProperties,
       LoanLibrary.Predicate[] calldata itemPredicates
   ) public override returns (uint256 loanId) {
       ...
       Side neededSide = isSelfOrApproved(borrowerData.borrower, msg.sender) ?
       Side.LEND : Side.BORROW;

       (bytes32 sighash, address externalSigner) = _recoverSignature(loanTerms, sig,
       sigProperties, neededSide, itemPredicates);
       ...
       loanCore.consumeNonce(externalSigner, sigProperties.nonce,
       sigProperties.maxUses);
>       loanId = _initialize(loanTerms, borrowerData, lender);
```

```
  function _initialize(
        LoanLibrary.LoanTerms calldata loanTerms,
        BorrowerData calldata borrowerData,
        address lender
    ) internal nonReentrant returns (uint256 loanId) {
        ...
        if (borrowerData.callbackData.length > 0) {
>           IExpressBorrow(borrowerData.borrower).executeOperation(msg.sender,
lender, loanTerms, borrowerFee, borrowerData.callbackData);
        }
```

However, it can be observed that `borrowerData.callbackData` isn't included in the signature

```
» (bytes32 sighash, address externalSigner) = _recoverSignature(loanTerms, sig,
sigProperties, neededSide, itemPredicates);
```

This becomes an issue when `neededSide` is `Side.BORROW` since the caller is the lender - the caller can modify `borrowerData.callbackData` or pass empty bytes, and the call would still pass. If the borrower's `executeOperation()` callback uses `callbackData`, it could function differently than expected.

Hence, the signature should include the callback data here so the counterparty can know if the callback will be executed.

**Recommendation:** Include `borrwerData.callbackData` in the signature:

```
function initializeLoan(
        LoanLibrary.LoanTerms calldata loanTerms,
        BorrowerData calldata borrowerData,
        address lender,
        Signature calldata sig,
        SigProperties calldata sigProperties,
        LoanLibrary.Predicate[] calldata itemPredicates
    ) public override returns (uint256 loanId) {
        ...

-       (bytes32 sighash, address externalSigner) = _recoverSignature(loanTerms, sig,
sigProperties, neededSide, itemPredicates);
+       (bytes32 sighash, address externalSigner) = _recoverSignature(loanTerms, sig,
sigProperties, neededSide, itemPredicates, borrowerData.callbackData);
        ...
        loanCore.consumeNonce(externalSigner, sigProperties.nonce,
        sigProperties.maxUses);
        loanId = _initialize(loanTerms, borrowerData, lender);
```

**Arcade:** Fixed in PR-118.

**Renascence:** The issue has been fixed as recommended.

#### [M-10] `migrateV3Loan()` will revert for lenders that are approved or use ERC-1271

**Context:** OriginationControllerMigrate.sol

**Description:** `migrateV3Loan()` checks that `externalSigner`, the address retrieved from the signature, is the same as the `lender` address:

```
    function migrateV3Loan(
        uint256 oldLoanId,
        LoanLibrary.LoanTerms calldata newTerms,
        address lender,
        Signature calldata sig,
        SigProperties calldata sigProperties,
        LoanLibrary.Predicate[] calldata itemPredicates
    ) external override whenNotPaused whenBorrowerReset {
          ...
»        if (externalSigner != lender) revert OCM_SideMismatch(externalSigner);
```

However, checking that `externalSigner` is the `lender` address will revert for lenders that:

1. Are smart wallet addresses that validate signatures through ERC-1271.
2. Approve other addresses to create offers on their behalf using `approve()`

As such, borrowers cannot migrate their V3 loans using loan offers in V4 if the lender is either of the above.

**Recommendation:** Consider validating lender signatures in `migrateV3Loan()` as such:

```
- (, address externalSigner) = _recoverSignature(newTerms, sig, sigProperties,
Side.LEND, itemPredicates);
+ (bytes32 sighash, address externalSigner) = _recoverSignature(newTerms, sig,
sigProperties, Side.LEND, itemPredicates);

  // revert if the signer is not the lender
- if (externalSigner != lender) revert OCM_SideMismatch(externalSigner);
+ if (!isSelfOrApproved(lender, externalSigner) &&
!OriginationLibrary.isApprovedForContract(lender, sig, sighash)) {
+     revert OCM_SideMismatch(externalSigner);
+ }
```

Additionally, the `externalSigner` address should be passed to `loanCore.consumeNonce()`, instead of `lender`:

```
  // consume v4 nonce
- loanCore.consumeNonce(lender, sigProperties.nonce, sigProperties.maxUses);
+ loanCore.consumeNonce(externalSigner, sigProperties.nonce, sigProperties.maxUses);
```

**Arcade:** Fixed in PR-111.

**Renascence:** The issue has been fixed as recommended.

#### [M-11] Stale `feeSnapshot` is used when rolling over a loan

**Context:**

- OriginationController.sol

- LoanCore.sol

**Description:** When rolling over a loan, instead of fetching the most recent fees through `FeeController.getOriginationFeeAmounts()` the function assigns the new loan fees by reusing `data.feeSnapshot` that was taken when the previous loan was created.

This means that even if the protocol's default, interest and principal fees were updated, the new loan created by `rollover()` would still use old fee values.

```
  function rollover(
        uint256 oldLoanId,
        address oldLender,
        address borrower,
        address lender,
        LoanLibrary.LoanTerms calldata terms,
        uint256 _settledAmount,
        uint256 _amountToOldLender,
        uint256 _amountToLender,
        uint256 _amountToBorrower,
        uint256 _interestAmount
    ) external override whenNotPaused onlyRole(ORIGINATOR_ROLE) nonReentrant returns
    (uint256 newLoanId) {
>        LoanLibrary.LoanData storage data = loans[oldLoanId];
        ...
        loans[newLoanId] = LoanLibrary.LoanData({
            state: LoanLibrary.LoanState.Active,
            startDate: uint64(block.timestamp),
            lastAccrualTimestamp: uint64(block.timestamp),
            terms: terms,
>            feeSnapshot: data.feeSnapshot,
            balance: terms.principal,
            interestAmountPaid: 0
        });
```

**Recommendation:** When rolling over a loan, the new loan's `feeSnapshot` should be fetched using `FeeController.getOriginationFeeAmounts()`.

```
## OriginationController.sol

 function _rollover(
        uint256 oldLoanId,
        LoanLibrary.LoanTerms calldata newTerms,
        address borrower,
        address lender
    ) internal nonReentrant returns (uint256 loanId) {
        ...
+        LoanLibrary.FeeSnapshot memory feeSnapshot =
FeeController.getOriginationFeeAmounts(newTerms.principal);

        loanId = loanCore.rollover(
            oldLoanId,
            oldLender,
            borrower,
            lender,
            newTerms,
            settledAmount,
            amounts.amountToOldLender,
            amounts.amountToLender,
            amounts.amountToBorrower,
            amounts.interestAmount,
+            feeSnapshot
        );
```

```
## LoanCore.sol

  function rollover(
        uint256 oldLoanId,
        address oldLender,
        address borrower,
        address lender,
        LoanLibrary.LoanTerms calldata terms,
        uint256 _settledAmount,
        uint256 _amountToOldLender,
        uint256 _amountToLender,
        uint256 _amountToBorrower,
        uint256 _interestAmount,
+        LoanLibrary.FeeSnapshot memory feeSnapshot
    ) external override whenNotPaused onlyRole(ORIGINATOR_ROLE) nonReentrant returns
    (uint256 newLoanId) {
        ...
        loans[newLoanId] = LoanLibrary.LoanData({
            state: LoanLibrary.LoanState.Active,
            startDate: uint64(block.timestamp),
            lastAccrualTimestamp: uint64(block.timestamp),
            terms: terms,
-            feeSnapshot: data.feeSnapshot,
+            feeSnapshot: feeSnapshot,
            balance: terms.principal,
            interestAmountPaid: 0
        });
```

**Arcade:** This is a design decision.

**Renascence:** The issue has been acknowledged.

### [M-12] `totalOwed` does not account for partial repayments

**Context:** RepaymentController.sol

**Description:** When claiming, the `totalOwed` calculation does not account for the newly introduced partial repayments.

```
    function claim(uint256 loanId) external override {
          ...
>         uint256 totalOwed = terms.principal + interest;
```

Assume that the borrower had repaid part of the loan.

Then, `terms.principal` would still be the full loan principal, but `interest` would be the remaining interest that has yet to be paid.

As a result, the lender will pay more/less claim fees, depending on the intended fee for claiming a defaulted loan.

For example, the lender could call `repay()` to repay his loan's currently owed interest. This would cause `interest` to become `0`, therefore the lender would avoid paying claim fees on interest.

**Recommendation:** If claim fees are meant to be calculated based on the entire loan's value (principal + total interest), add the amount of interest already paid to `totalOwed`:

```
- uint256 totalOwed = terms.principal + interest;
+ uint256 totalOwed = terms.principal + interest + data.interestAmountPaid
```

**Arcade:** Fixed in PR-112.

**Renascence:** The issue has been fixed as recommended.

### [M-13] `lenderFee` and `interestFee` may not be collected from new lenders

**Context:** OriginationCalculator.sol

**Description:** In `OriginationCalculator. rolloverAmounts()`, under the following two conditions:

1. `lender == oldLender`
2. `borrowerOwedForNewLoan < repayAmount < amountFromLender`

The `lenderFee` and `interestFee` will not be collected from the new lender and will not be paid. Additionally, a portion of `borrowFee` isn't paid as `newPrincipalAmount - repayAmount` isn't collected from the new lender.

When both conditions listed above are true:

- `amounts.leftoverPrincipal` will be `0`, so no amount is collected from the new lender
- `amountToLender` will be `0`, so no fees are subtracted from the amount sent to the new lender

As such, the ”costs” to the new lender aren't accounted for or collected. As a result, the protocol and old/new loan affiliates will lose out on fees.

For example, assuming `payableCurrency` is USDC:

```
repayAmount = 80
newPrincipalAmount = 100
borrowerFee = lenderFee = interestFee = 30

borrowerOwedForNewLoan = newPrincipalAmount - borrowerFee = 100 - 30 = 70
amounts.amountFromLender = newPrincipalAmount + lenderFee + interestFee = 100 + 30 +
30 = 160

amounts.needFromBorrower = repayAmount - borrowerOwedForNewLoan = 80 - 70 = 10
```

Only 10 USDC was collected from the borrower, and there were no other transfers. Therefore, only 10 USDC will be accrued as the `borrowerFee`.

The missing funds are:

- `lenderFee` and `interestFee`, which are 30 USDC each.
- A portion of `borrowerFee`, more specifically, 20 USDC.

The sum of these missing funds is equal to `amounts.amountFromLender - repayAmount`, which was supposed to be collected from the new lender.

**Additional Note:** The current V3 `OriginationController` live contract also contains the same bug, but all fees are currently set to `0`.

**Recommendation:** When `borrowerOwedForNewLoan < repayAmount < amountFromLender`, set `amounts.leftoverPrincipal` as `amounts.amountFromLender - repayAmount`:

```
  if (repayAmount > borrowerOwedForNewLoan) {
+     if (repayAmount < amounts.amountFromLender) {
+         amounts.leftoverPrincipal = amounts.amountFromLender - repayAmount;
+     }

      // amount to collect from borrower
      unchecked {
          amounts.needFromBorrower = repayAmount - borrowerOwedForNewLoan;
      }
  } else {
```

This will collect the missing funds from the lender when `lender == oldLender`.

Additionally, modify `_rollover()` to collect `leftoverPrincipal` when `needFromBorrower > 0`, similar to the logic in `OriginationControllerMigrate._migrate()`:

```
    // Collect funds based on settle amounts and total them
    uint256 settledAmount;
    if (lender != oldLender) {
        // If new lender, take new principal from new lender
        payableCurrency.safeTransferFrom(lender, address(this),
        amounts.amountFromLender);
        settledAmount += amounts.amountFromLender;
-   }
+   } else if (amounts.leftoverPrincipal > 0) {
+       payableCurrency.safeTransferFrom(lender, address(this),
amounts.leftoverPrincipal);
+       settledAmount += amounts.leftoverPrincipal;
+   }

    if (amounts.needFromBorrower > 0) {
        // Borrower owes from old loan
        payableCurrency.safeTransferFrom(borrower, address(this),
        amounts.needFromBorrower);
        settledAmount += amounts.needFromBorrower;
-   } else if (amounts.leftoverPrincipal > 0 && lender == oldLender) {
-       // If same lender, and new amount from lender is greater than old loan repayment
amount,
-       // take the difference from the lender
-       payableCurrency.safeTransferFrom(lender, address(this),
amounts.leftoverPrincipal);
-       settledAmount += amounts.leftoverPrincipal;
    }
```

**Arcade:** Fixed in PR-113.

**Renascence:** The issue has been fixed as recommended.

### [M-14] Reusable signatures can be repeatedly rolled over with `rolloverLoan()` to incur extra fees

**Context:** OriginationCalculator.sol#L133-L138

**Description:** If a user with a reusable signature enters a loan, the other party can just repeatedly call `rolloverLoan()` with the same signature to consume all their signatures. Assuming that all fees are at 0%, this will just incur gas costs for the attacker.

The more serious impact is if:

- The victim granted infinite approval of `payableCurrency` to the `OriginationController` contract.
- Fees are not at 0%.

By repeatedly calling `rolloverLoan()`, an attacker can cause the other party to keep incurring fees.

Fees for rollovers are calculated in `_calculateRolloverAmounts()`:

```
// Calculate amount to be sent to borrower for new loan minus rollover fees
uint256 borrowerFee = (newPrincipalAmount * feeData.borrowerRolloverFee) /
Constants.BASIS_POINTS_DENOMINATOR;

// Calculate amount to be collected from the lender for new loan plus rollover fees
uint256 interestFee = (interest * oldLoanData.feeSnapshot.lenderInterestFee) /
Constants.BASIS_POINTS_DENOMINATOR;
uint256 lenderFee = (newPrincipalAmount * feeData.lenderRolloverFee) /
Constants.BASIS_POINTS_DENOMINATOR;
```

Assuming that a principal fee is implemented for `rolloverLoan()`, for each rollover:

- Borrower will incur rollover fees.
- Lender will incur rollover and principal fees. There are no interest fees if the attacker calls `rolloverLoan()` repeatedly, since no time has passed since loan creation.

An example where repeatedly calling `rolloverLoan()` to grief the opposite party is likely:

- Rollover fees are at 0%, but principal fees are non-zero.
- Lender signs a reusable signature and grants infinite approval to the `OriginationController` contract.
- After the lender entires a loan, the borrower repeatedly calls `rolloverLoan()`:
  - Lender will pay principal fees for each rollover.
  - However, the borrower doesn't pay any fees.

**Recommendation:** Ensure that rollover fees are always greater than principal fees. More specifically, `FL_03`, which is the `borrowerRolloverFee`, should always be more than `FL_07`, the `lenderPrincipalFee`.

This ensures that it is never economically viable to perform the attack described above.

**Arcade:** Code refactoring in PR-108 addresses the issue.

**Renascence:** The issue has been fixed.

## 4.3 Low Risk

**[L-1] Missing `isAllowedCollateral()` inside `OriginationControllerMigrate_validateV3Migration()`**

**Context:** OriginationControllerMigrate.sol

**Description:** Inside `OriginationControllerMigrate_validateV3Migration()` is no check if the `newLoanTerms.collateralAddress` is allowed collateral in V4.

This means a loan with allowed collateral in V3 can be migrated to V4, even if the collateral is not allowed in V4.

**Recommendation:**

```
+ if (!originationConfiguration.isAllowedCollateral(newLoanTerms.collateralAddress))
revert OCM_InvalidCollateral(newLoanTerms.collateralAddress);
```

Also, add the error in `Lending.sol` and Import it in `OriginationControllerMigrate`

```
+ error OCM_InvalidCollateral(address collateralAddress);
```

An alternative solution could be to utilize `OriginationConfiguration.validateLoanTerms()` to check the correctness of the new `LoanTerms`.

**Arcade:** Code is also refactored to validate the terms with `OriginationConfiguration.validateLoanTerms()`, PR-114.

**Renascence:** The issue has been fixed as recommended.

**[L-2] `RefinanceController._validateRefinance()` missing `MIN_LOAN_DURATION` check**

**Context:** RefinanceController.sol

**Description:** In `RefinanceController._validateRefinance()`, there is a missing check that the loan duration is not less than `MIN_LOAN_DURATION`. If `refinanceLoan()` is called close to or after `oldDueDate`, users can refinance with a duration less than `MIN_LOAN_DURATION`.

**Recommendation:** Check that the new loan's duration is not less than `MIN_LOAN_DURATION`:

```
@@ -120,6 +120,7 @@ contract RefinanceController is IRefinanceController,
OriginationCalculator, Ree
         uint256 newDueDate = block.timestamp + newTerms.durationSecs;
         if (
             newDueDate < oldDueDate ||
+            newTerms.durationSecs < Constants.MIN_LOAN_DURATION ||
             newTerms.durationSecs > Constants.MAX_LOAN_DURATION
         ) revert REFI_LoanDuration(oldDueDate, newDueDate);
```

**Arcade:** Fixed in PR-115.

**Renascence:** The issue has been fixed as recommended.

### [L-3] `RefinanceController._validateRefinance()` missing minimal principal check

**Context:** RefinanceController.sol

**Description:** `OriginationConfiguration.validateLoanTerms()` checks if the supplied term's principal is above the allowed minimal principal for the supplied currency.

The validation function is used in `OriginationController.initializeLoan()` and `OriginationController.rolloverLoan()`.

```
function validateLoanTerms(LoanLibrary.LoanTerms memory terms) public view {
    // validate payable currency
    if (!isAllowedCurrency(terms.payableCurrency)) revert
    OCC_InvalidCurrency(terms.payableCurrency);

    // principal must be greater than or equal to the configured minimum
    if (terms.principal < getMinPrincipal(terms.payableCurrency)) revert
    OCC_PrincipalTooLow(terms.principal);

    // code ...
}
```

However, `RefinanceController._validateRefinance()` does not do a minimal principal check, which violates the assumption that loans should have a principal over the set minimum.

**Recommendation:**

```
@@ -139,7 +139,9 @@ contract RefinanceController is IRefinanceController,
OriginationCalculator, Ree
             oldLoanData.terms.payableCurrency,
             newTerms.payableCurrency
         );
+
+        if (newTerms.principal <
originationConfig.getMinPrincipal(newTerms.payableCurrency)) revert
REFI_PrincipalTooLow(newTerms.principal)
+
```

**Arcade:** Fixed in PR-115.

**Renascence:** The issue has been fixed as recommended.

### [L-4] Lender notes should not be traded on the secondary market

**Context:**

- RepaymentController.sol#L169-L186
- LoanCore.sol#L298-L302

**Description:** Since borrower and lender notes are ERC-721 tokens, they can be bought/sold on secondary markets such as OpenSea.

However, the current design of the protocol allows sellers to front-run transactions and decrease the value of lender notes.

For example:

- Lenders can call `redeemNote()` to claim repayments held in note receipts.
- Borrower can call `forceRepay()` and fully repay the loan, leaves a worthless lender note behind since the collateral is withdrawn.

Note that the lender note is not burned when `forceRepay()` is called, even on full repayments:

```
if (loans[loanId].state == LoanLibrary.LoanState.Repaid) {
    // if loan is completely repaid
    // burn BorrowerNote, DO NOT burn LenderNote until receipt is redeemed
    address borrower = borrowerNote.ownerOf(loanId);
    borrowerNote.burn(loanId);
```

As such, if lender notes were sold on the secondary market, there is a risk of the seller front-running a buyer's transaction to remove all value from the loan, right before the buyer's transaction is executed.

**Recommendation:** Consider documenting that users should not trade lender notes on the secondary market.

**Arcade:** Documented in `README`, PR-116.

**Renascence:** The issue has been fixed as recommended.

#### [L-5] Lack of state update in `LoanCore.rollover()` can lead to inflated effective APR

**Context:** LoanCore.sol

**Description:** When calling `LoanCore.rollover()`, `data.lastAccrualTimestamp` isn't updated to `block.timestamp`.

This is inconsistent with `_handleRepay()`, which updates `data.lastAccrualTimestamp` regardless of whether the loan gets fully repaid or not:

```
loans[loanId].interestAmountPaid += _interestAmount;
loans[loanId].balance -= _paymentToPrincipal;
loans[loanId].lastAccrualTimestamp = uint64(block.timestamp);
```

This will cause `getCloseEffectiveInterestRate()` to return an inflated effective APR when called after `rollover()`.

**Recommendation:** Update `data.lastAccrualTimestamp` in `rollover()`:

```
  // State change for old loan
  data.state = LoanLibrary.LoanState.Repaid;
  data.balance = 0;
  data.interestAmountPaid += _interestAmount;
+ data.lastAccrualTimestamp = uint64(block.timestamp);
```

**Arcade:** Fixed in PR-117.

**Renascence:** The issue has been fixed as recommended.

## 4.4 Informational

**[I-01] Use a `MAX_LENGTH` constant for array length checks in `OriginationController.sol`**

**Context:**

- OriginationConfiguration.sol#L140
- OriginationConfiguration.sol#L170
- OriginationConfiguration.sol#L200

**Description:** The following lines in `OriginationController.sol` check that input arrays do not have a length greater than `50`:

```
if (verifiers.length > 50) revert OCC_ArrayTooManyElements();
```

```
if (tokens.length > 50) revert OCC_ArrayTooManyElements();
```

```
if (tokens.length > 50) revert OCC_ArrayTooManyElements();
```

It is best practice to use a constant instead of hardcoded value in multiple places of the code.

**Recommendation:** Consider declaring a `MAX_LENGTH = 50` constant and using it in the checks above.

**Arcade:** Fixed in PR-99.

**Renascence:** The issue has been fixed as recommended.

**[I-02] Errors in comments**

**Context:**

- OriginationConfiguration.sol#L38
- LoanCore.sol#L777-L780
- OriginationControllerMigrate.sol#L138-L139

**Description:** Collateral can only be ERC-721; ERC-1155 tokens are not allowed collateral:

OriginationConfiguration.sol#L38

```
- /// @notice Mapping from ERC721 or ERC1155 token address to boolean indicating
allowed collateral types
+ /// @notice Mapping from ERC721 token address to boolean indicating allowed
collateral types
```

`claim()` doesn't have the `whenNotPaused` modifier, so defaults can be claimed after shutdown:

LoanCore.sol#L777-L780

```
/**
 * @notice Shuts down the contract, callable by a designated role. Irreversible.
 *         When the contract is shutdown, loans can only be repaid.
 *         New loans cannot be started, defaults cannot be claimed,
```

Wrong version in the NatSpec of `OriginationControllerMigrate._validateV3Migration()`. The comment `sourceLoanTerms` should refer to V3 loan terms and `newLoanTerms` should refer to V4 loan terms:

OriginationControllerMigrate.sol#L138-L139

```
@@ -135,8 +136,8 @@ contract OriginationControllerMigrate is IMigrationBase,
OriginationController,
      * @dev All whitelisted payable currencies and collateral state on v3 must also
      be set to the
      *      same values on v4.
      *
-     * @param sourceLoanTerms           The terms of the V2 loan.
-     * @param newLoanTerms              The terms of the V3 loan.
+     * @param sourceLoanTerms           The terms of the V3 loan.
+     * @param newLoanTerms              The terms of the V4 loan.
      * @param borrowerNoteId            The ID of the borrowerNote for the old loan.
      */
```

**Recommendation:** Consider amending the comments listed above.

**Arcade:** Fixed in PR-99.

**Renascence:** The issue has been fixed as recommended.

#### [I-03] Overriding `_unpause()` in `LoanCore.sol` is unnecessary

**Context:** LoanCore.sol#L960-L965

**Description:** The `LoanCore` contract overrides `_unpause()` to prevent the contract from unpausing once `shutdown()` is called:

```
/**
 * @dev Blocks the contract from unpausing once paused.
 */
function _unpause() internal override whenPaused {
    revert LC_Shutdown();
}
```

This override is not needed as the contract doesn't implement a function that calls `_unpause()`.

**Recommendation:** Consider removing the `_unpause()` function.

**Arcade:** Fixed in PR-99.

**Renascence:** The issue has been fixed as recommended.

### [I-04] `aprMinimumScaled` calculation in `_validateRefinance()` can be simplified

**Context:** RefinanceController.sol#L110-L112

**Description:** The following statement in `RefinanceController. _validateRefinance()`:

```
// new interest rate APR must be lower than old interest rate by minimum
uint256 aprMinimumScaled = oldLoanData.terms.interestRate *
Constants.BASIS_POINTS_DENOMINATOR -
    (oldLoanData.terms.interestRate * MINIMUM_INTEREST_CHANGE);
```

can be simplified to:

```
uint256 aprMinimumScaled = oldLoanData.terms.interestRate *
(Constants.BASIS_POINTS_DENOMINATOR - MINIMUM_INTEREST_CHANGE);
```

**Recommendation:** Consider modifying the statement as suggested above.

**Arcade:** Fixed in PR-99.

**Renascence:** The issue has been fixed as recommended.

### [I-05] Checking `data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE` in `_prepareRepay()` is redundant

**Context:** RepaymentController.sol#L212-L214

**Description:** `RepaymentController.sol._prepareRepay()` contains the following loan state checks:

```
// loan state checks
if (data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE) revert
RC_CannotDereference(loanId);
if (data.state != LoanLibrary.LoanState.Active) revert RC_InvalidState(data.state);
```

However, the `data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE` check is redundant since it is covered by the `data.state != LoanLibrary.LoanState.Active` check.

**Recommendation:** Consider removing the `data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE` check:

```
  // loan state checks
- if (data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE) revert
RC_CannotDereference(loanId);
  if (data.state != LoanLibrary.LoanState.Active) revert RC_InvalidState(data.state);
```

**Arcade:** Fixed in PR-100.

**Renascence:** The issue has been fixed as recommended.

#### [I-06] `paymentToPrincipal` calculation can be unchecked in `_prepayRepay()`

**Context:** RepaymentController.sol#L226-L230

**Description:** `RepaymentController._prepayRepay()` contains the following logic:

```
// make sure that repayment amount is greater than interest due
if (amount < interestAmount) revert RC_InvalidRepayment(amount, interestAmount);

// calculate the amount of the repayment that goes to the principal
paymentToPrincipal = amount - interestAmount;
```

The calculation of `paymentToPrincipal` can be left unchecked since `amount - interestAmount` can never overflow, due to the check above.

**Recommendation:** Consider making the calculation of `paymentToPrincipal` unchecked:

```
  // calculate the amount of the repayment that goes to the principal
- paymentToPrincipal = amount - interestAmount;
+ unchecked { paymentToPrincipal = amount - interestAmount; }
```

**Arcade:** Fixed in PR-101.

**Renascence:** The issue has been fixed as recommended.

#### [I-07] Design improvement in fee query

**Context:** OriginationControllerMigrate.sol

**Description:** In `OriginationControllerMigrate.migrateV3Loan()`, there are currently two queries to `FeeController.getOriginationFeeAmounts()` for the same fees, once in `OriginationControllerMigrate._migrate()` and once in `OriginationControllerMigrate._initializeMigrationLoan()`.

**Recommendation:** Consider refactoring the code to query `FeeController.getOriginationFeeAmounts()` only once.

**Arcade:** Addressed in PR-96.

**Renascence:** The issue has been fixed as recommended.

#### [I-08] Inconsistency in `RepaymentController.claim()`

**Context:** RepaymentController.sol

**Description:** The function `RepaymentController.claim()` doesn't contain a `data.state != LoanLibrary.LoanState.Active` check. Although this check is present in `LoanCore.claim()`, consider adding the check in `RepaymentController.claim()` to be consistent with other functions that contain this check in `RepaymentController.sol` and `LoanCore.sol`.

**Recommendation:**

```
@@ -142,7 +142,7 @@ contract RepaymentController is IRepaymentController,
InterestCalculator, FeeLoo
      */
     function claim(uint256 loanId) external override {
         LoanLibrary.LoanData memory data = loanCore.getLoan(loanId);
-        if (data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE) revert
RC_CannotDereference(loanId);
+        if (data.state != LoanLibrary.LoanState.Active) revert
RC_InvalidState(loanId);
```

**Arcade:** Fixed in PR-100.

**Renascence:** The issue has been fixed as recommended.

### [I-09] `OriginationLibrary.isApprovedForContract()` should decode `result` as `bytes32` to avoid unexpected reverts in future use.

**Context:** OriginationLibrary.sol

**Description:** Decoding to `bytes4`, `abi.decode(result, (bytes4))`, in `OriginationLibrary.isApprovedForContract()` will cause a revert if the data in `result` is more than 4 bytes long. An example would be if `isValidSignature()`, or the fallback function at `target`, returned `bytes5` instead of `bytes4`.

```
function isApprovedForContract(
    address target,
    IOriginationController.Signature memory sig,
    bytes32 sighash
) public view returns (bool) {
    // code ...
    // Convert sig struct to bytes
    (bool success, bytes memory result) = target.staticcall(
        abi.encodeWithSelector(IERC1271.isValidSignature.selector, sighash, signature)
    );
    return (success && result.length == 32 && abi.decode(result, (bytes4)) ==
    IERC1271.isValidSignature.selector);
}
```

This is why OpenZeppelin's `SignatureChecker.sol` decodes to a `bytes32` instead:

SignatureChecker.sol#L44-L46

```
return (success &&
    result.length >= 32 &&
    abi.decode(result, (bytes32)) == bytes32(IERC1271.isValidSignature.selector));
```

There isn't any impact on the current codebase as `_validateCounterparties()` also reverts when `isApprovedForContract()` returns `false`.

However, this is something to worth keeping in mind if any future functionality uses `isApprovedForContract()` and isn't meant to revert when the signature check fails.

**Recommendation:** Consider decoding to a `bytes32` instead, similar to OpenZeppelin's implementation:

```
@@ -126,6 +126,6 @@ library OriginationLibrary {
         (bool success, bytes memory result) = target.staticcall(
             abi.encodeWithSelector(IERC1271.isValidSignature.selector, sighash,
             signature)
         );
-        return (success && result.length == 32 && abi.decode(result, (bytes4)) ==
IERC1271.isValidSignature.selector);
+        return (success && result.length == 32 && abi.decode(result, (bytes32)) ==
bytes32(IERC1271.isValidSignature.selector));
     }
 }
```

**Arcade:** Fixed in PR-102.

**Renascence:** The issue has been fixed as recommended.

#### [I-10] `setAffiliateSplits()` lacks `whenNotPaused` and `nonReentrant` modifiers

**Context:** LoanCore.sol **Description:** The `whenNotPaused` and `nonReentrant` modifiers are missing from `setAffiliateSplits()`:

```
    function setAffiliateSplits(
        bytes32[] calldata codes,
        AffiliateSplit[] calldata splits
>    ) external override onlyRole(AFFILIATE_MANAGER_ROLE) {
```

Consequently, permissioned roles can re-enter `setAffiliateSplits()`, and the function can be called after the contract is shut down with `shutdown()`.

While `setAffiliateSplit()` can only be called by the `AFFILIATE_MANAGER_ROLE`, it is best to apply these modifiers to all functions consistently.

**Recommendation:**

```
  function setAffiliateSplits(
      bytes32[] calldata codes,
      AffiliateSplit[] calldata splits
- ) external override onlyRole(AFFILIATE_MANAGER_ROLE) {
+ ) external override nonReentrant whenNotPaused onlyRole(AFFILIATE_MANAGER_ROLE) {
```

**Arcade:** Fixed in PR-103.

**Renascence:** The issue has been fixed as recommended.

#### [I-11] `amounts.amountFromLender > 0` check in `OriginationControllerMigrate._migrate()` is incorrect

**Context:** OriginationControllerMigrate.sol#L234-L236

**Description:** `_migrate()` handles transfers of `leftoverPrincipal` from the lender as such:

```
if (amounts.amountFromLender > 0) {
    payableCurrency.safeTransferFrom(lender, address(this),
    amounts.leftoverPrincipal);
}
```

However, the check here should be `amounts.leftoverPrincipal > 0` instead as `amounts.amountFromLender` will always be greater than `0`.

As a result, `_migrate()` will perform unnecessary transfers with 0 amount.

**Recommendation:** Modify the check to `amounts.leftoverPrincipal > 0` instead:

```
- if (amounts.amountFromLender > 0) {
+ if (amounts.leftoverPrincipal > 0) {
      payableCurrency.safeTransferFrom(lender, address(this),
      amounts.leftoverPrincipal);
  }
```

**Arcade:** Fixed in PR-104.

**Renascence:** The issue has been fixed as recommended.

#### [I-12] Interest calculation in `getProratedInterestAmount()` can round down to 0 for tokens with low decimals

**Context:** InterestCalculator.sol#L57-L58

**Description:** `getProratedInterestAmount()` calculates the amount of interest due for a loan as such:

```
interestAmountDue = balance * timeSinceLastPayment * interestRate
    / (Constants.BASIS_POINTS_DENOMINATOR * Constants.SECONDS_IN_YEAR);
```

Note that the calculation here can round down to `0` in extreme scenarios.

For example, assume the following:

- `payableCurrency` is GUSD, which has 2 decimals.
- `timeSinceLastPayment = 3600`, which is 1 hour.
- `interestRate = 500`, which is 5% APR.

If `balance` is less than `175200` , `interestAmountDue` will round down to `0`. This means that borrowers can avoid accruing interest by repaying every hour when their principal is less than $1752.

However, due to gas fees on mainnet, this is most likely impossible to exploit in practice.

**Recommendation:** Consider documenting this behavior in the comments. Additionally, avoid whitelisting tokens with extremely low decimals for `payableCurrency`.

**Arcade:** Fixed in PR-104.

**Renascence:** The issue has been fixed as recommended.

#### [I-13] Stale data after claiming a loan

**Context:** LoanCore.sol

**Description:** When calling `LoanCore.claim()`, `loans[loanId].balance` and `loans[loanId].lastAccrualTimestamp` are not updated.

However, there isn't any noticeable impact since all functions cannot be called for that have a state of `LoanState.Defaulted`.

**Recommendation:** To have consistent state handling, consider updating `loans[loanId].balance` and `loans[loanId].lastAccrualTimestamp` as well:

```
    // State changes and cleanup
    loans[loanId].state = LoanLibrary.LoanState.Defaulted;
+   loans[loanId].balance = 0;
+   loans[loanId].lastAccrualTimestamp = uint64(block.timestamp);

    collateralInUse[keccak256(abi.encode(data.terms.collateralAddress,
    data.terms.collateralId))] = false;
```

**Arcade:** Fixed in PR-104.

**Renascence:** The issue has been fixed as recommended.