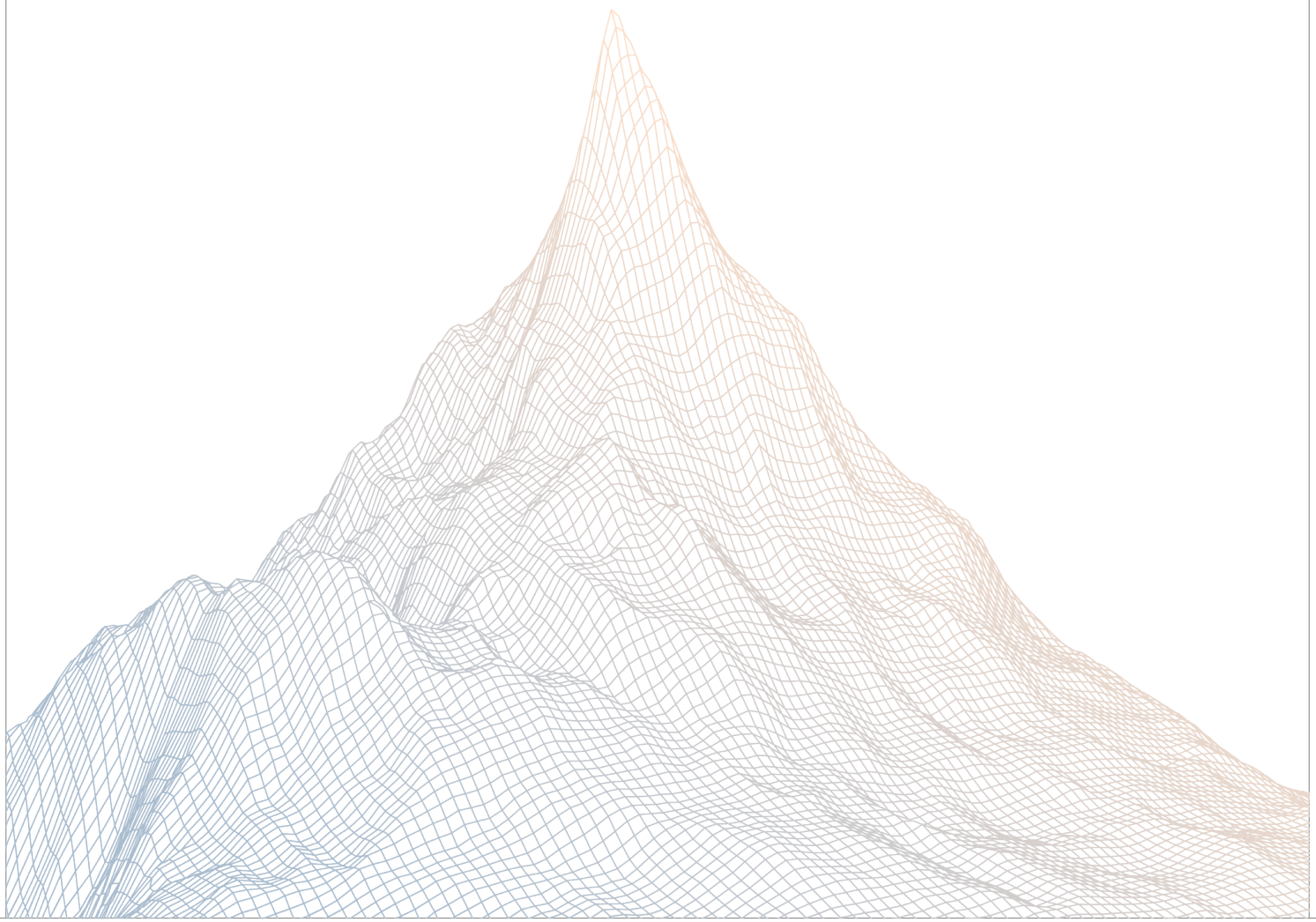
Zenith

Gondi

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: August 19th to 21st, 2026

AUDITED BY: oakcobalt
said

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	`b5f56d7b153baec07f6fb874cfd20a3ab28af096`
Files	`src/**/*`

Target	Mitigation Review
Repository	https://github.com/pixeldaogg/florida-contracts
Commit Hash	`47beaf3408b47c3d48d053aa1b298067a7fe4659`
Files	`src/**/*`

2.3 Audit Timeline

August 19, 2026	Audit start
August 21, 2026	Audit end
August 24, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	1
Informational	8
Total Issues	**11**

3

Findings Summary

ID	Description	Status
M-1	ETH-funded swapAndExecute DOS atomic sell (OOS)	Resolved
M-2	Borrower can set lenderRefinanceDisabled on a pool-backed loan the pool never agreed to	Resolved
L-1	The foreclosure exit charges no protocol fee	Acknowledged
I-1	Refinance callback is passed the loan being replaced, not the loan just initiated	Resolved
I-2	lenderRefinanceDisabled blocks borrower-initiated renegotiation and tranche additions, not only lender refinances	Resolved
I-3	The lenderRefinanceDisabled transfer guarantee holds at settlement but is not a per-transfer property	Resolved
I-4	One pool can originate loans on both v3.1 and v3.2	Resolved
I-5	v3.2 MultiSourceLoan ABI cannot share v3.1 companions, and no YAML deploys replacements	Resolved
I-6	Unused LoanOffer getInterest method	Resolved
I-7	Integrations routing principal to a borrower-controlled address break on flagged refinance	Resolved
I-8	The two new features together give a lender a targeted, uncontested foreclosure-to-own right on a specific NFT	Resolved

4

Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] ETH-funded swapAndExecute DOS atomic sell (OOS)

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

Target:

- src/lib/callbacks/PurchaseBundler.sol#L424-L492

Description:

Payable `swapAndExecute` collects ETH as `msg.value`, optionally sends `address(this).balance` to the Uniswap router, then `delegatecall`s allowlisted self-targets including non-payable `sell`. `DELEGATECALL` keeps the current `CALLVALUE` and has no value argument, so `sell`'s Solidity non-payable prologue reverts when outer `msg.value` is non-zero even after the hop drains the contract's ETH. The low-level call returns `success = false` and `swapAndExecute` reverts. The intended swapAndExecute -> sell atomic flow doesn't work when an earlier swap needs ETH.

```
// src/lib/callbacks/PurchaseBundler.sol
▷ function sell(bytes[] calldata executionData, bytes[] calldata swapData)
    external nonReentrant whenNotPaused {
 ...
 }

 function executeSell(...)
    external payable nonReentrant whenNotPaused _storeMsgSender {
 ...
 }
```

```
// src/lib/callbacks/PurchaseBundler.sol
 (bool swapSuccess,) = address(_uniswapRouter).call{value:
    address(this).balance}(args.swapData);
 ...
 if (args.target == address(this)) {
```

```
 bytes4 sel = bytes4(args.executionCalldata[:4]);
 if (
▷ args.executionCalldata.length <= 4 || sel ≠ this.buy.selector && sel ≠
    this.sell.selector
 && sel ≠ this.executeSell.selector
 ) {
 revert InvalidExecutionData();
 }
 assembly {
 tstore(_SKIP_REENTRANCY_TOFFSET, 1)
 }
▷ (bool success,) = address(this).delegatecall(args.executionCalldata);
 if (!success) {
 revert InvalidCallbackError();
 }
```

A caller sends `swapAndExecute{value: v}` with `v > 0`, `args.target = address(this)`, and `executionCalldata` encoding `sell(...)`. After the optional swap, the inner `delegatecall` fails.

Impacts:

- Atomic self-revert of that ETH swap-then-sell composition; ETH never leaves the user.

Recommendations:

(1) Either remove `sell` from the self-target allowlist if ETH-bearing sales should stay on payable `executeSell`. (2) Else If `swapAndExecute` must still compose into `sell` after an ETH hop, dispatch `sell` with `address(this).call{value: 0}(...)` and keep `delegatecall` for payable `buy` and `executeSell`.

For example:

```
// src/lib/callbacks/PurchaseBundler.sol
 if (args.target == address(this)) {
 bytes4 sel = bytes4(args.executionCalldata[:4]);
 if (
 args.executionCalldata.length <= 4 || sel ≠ this.buy.selector && sel ≠
    this.sell.selector
 && sel ≠ this.executeSell.selector
 ) {
 revert InvalidExecutionData();
 }
 assembly {
 tstore(_SKIP_REENTRANCY_TOFFSET, 1)
 }
```

```
bool success;
if (sel == this.sell.selector) {
(success,) = address(this).call{value: 0}(args.executionCalldata);
} else {
(success,) = address(this).delegatecall(args.executionCalldata);
}
if (!success) {
revert InvalidCallbackError();
}
}
```

Gondi: Resolved with @4759852aec ...

Zenith: Verified.

[M-2] Borrower can set `lenderRefinanceDisabled` on a pool-backed loan the pool never agreed to

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

Target:

- PoolOfferHandler.sol#L138-L151
- ManagedPoolOfferHandler.sol#L231-L275
- MultiSourceLoan.sol#L1015-L1028

Description:

`lenderRefinanceDisabled` is meant to be the lender's choice. For a normal lender the field sits inside the offer hash the lender signs, so nobody else can set it. Pool offers work differently. When the lender is a registered `ILoanManager`, `_validateOfferExecution` skips the signature check and hands the offer to the pool's `validateOffer` instead:

```
if (lender.code.length != 0 && getLoanManagerRegistry.isLoanManager(lender))
    {
 ILoanManager(lender).validateOffer(_tokenId, abi.encode(_offerExecution),
    _feeFraction);
} else {
 _checkSignature(lender, offer.hash(), _lenderOfferSignature);
}
```

The borrower is the one who builds the entire `LoanOffer` struct for a pool loan, and neither `PoolOfferHandler.validateOffer` nor `ManagedPoolOfferHandler.validateOffer` ever reads `lenderRefinanceDisabled`:

```
function validateOffer(uint256, bytes calldata _offer)
    external view override returns (uint256, uint256) {
 IMultiSourceLoan.OfferExecution memory offerExecution = abi.decode(_offer,
    (IMultiSourceLoan.OfferExecution));
 if (offerExecution.offer.validators.length != 0) {
 revert InvalidOfferError();
 }
 uint256 apr = _terms[
   offerExecution.offer.nftCollateralAddress][
```

```
    offerExecution.offer.duration][
    offerExecution.offer.maxSeniorRepayment][
    offerExecution.offer.principalAmount];
  if (offerExecution.offer.aprBps < apr || apr == 0) {
  revert InvalidAprError();
  }
  // @audit offer.lenderRefinanceDisabled is never checked - borrower sets it
     freely
  return (offerExecution.amount, offerExecution.offer.aprBps);
}
```

The borrower-set flag then flows straight into the stored loan, where `_baseRenegotiationChecks` enforces it for the rest of the loan's life:

```
if (offer.lenderRefinanceDisabled) {
 if (totalOffers > 1) {
 revert LenderRefinanceDisabledError();
 }
 lenderRefinanceDisabled = true; // written onto the Loan
}
```

Once the loan carries the flag, `refinanceFull`, `refinancePartial`, and `addNewTranche` all revert, and `refinanceFromLoanExecutionData` is restricted to `principalReceiver == borrower`. The pool can no longer be refinanced out early by a competing better offer, and cannot exit the position before maturity or liquidation. The pool prices and books its loans expecting them to be refinanceable, a borrower can remove that property unilaterally.

Consider this scenario:

1. A pool advertises terms for a collection (say, 0.5 ETH at 30 days)
2. The borrower crafts a single pool `OfferExecution` matching those terms and sets `lenderRefinanceDisabled = true`
3. The pool accepts it (only balance, apr, and terms are checked) and funds the loan
4. The loan is stored as protected and single-tranche, with the pool as the lender
5. No competing lender can refinance the pool out, and the pool cannot roll or exit the position, its capital is locked until the borrower repays or the loan defaults and is liquidated

Repeated across many loans, this lets a borrower (or a competing pool operator) pin a meaningful slice of a pool's liquidity into non-refinanceable positions, degrading capital efficiency and withdrawal-queue servicing for LPs.

Recommendations:

Reject the flag on the pool path, where the pool cannot have consented to it.

Gondi: Resolved with PR-630.

Zenith: Verified.

4.2 Low Risk

A total of 1 low risk findings were identified.

[L-1] The foreclosure exit charges no protocol fee

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- LiquidationHandler.sol#L100-L105
- MultiSourceLoan.sol#L958-L976

Description:

Foreclosure transfers the NFT to the lender and touches neither `LiquidationDistributor` nor `_processRepayments`:

```
if (_canClaim) {
 ERC721(_loan.nftCollateralAddress)
 .transferFrom(address(this), _loan.tranche[0].lender,
    _loan.nftCollateralTokenId);
 emit LoanForeclosed(_loanId);
 liquidated = true;
}
```

The protocol charges no fee on the lender's in-kind gain, and the auction trigger fee is avoided. Repayment charges a protocol fee on interest, and the auction path charges its trigger fee, but foreclosure charges nothing. By making foreclosure selectable and guaranteed, `lenderRefinanceDisabled` lets a lender route a coveted-NFT loan into the one default exit that returns no revenue to the protocol. The fee-free foreclosure is pre-existing, the flag turns it into a reliable choice.

Recommendations:

If foreclosure gains should be fee-bearing, charge a protocol fee on the foreclosure branch, for example on value above the debt. Otherwise record the revenue tradeoff as accepted.

Gondi: Acknowledged. we will keep as is, we don't charge foreclosure.

4.3 Informational

A total of 8 informational findings were identified.

[I-1] Refinance callback is passed the loan being replaced, not the loan just initiated

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

Target:

- src/lib/loans/MultiSourceLoan.sol#L363-L377

Description:

`refinanceFromLoanExecutionData` builds a new `loan` and `totalFee` from incoming offers, then calls `handleAfterPrincipalTransferCallback` with existing `_loan` and that new fee. `emitLoan` passes the newly built `loan` into the same hook. `ILoanCallback.afterPrincipalTransfer` documents `_loan` as the loan just initiated.

```
// src/lib/loans/MultiSourceLoan.sol
 if (_hasCallback(executionData.callbackData)) {
 handleAfterPrincipalTransferCallback(loan, msg.sender,
    executionData.callbackData, totalFee); //@audit emitLoan
 }
```

```
// src/lib/loans/MultiSourceLoan.sol
 (uint256 newLoanId, uint256[] memory offerIds, Loan memory loan,
    uint256 totalFee) = _processOffersFromExecutionData(
 ...
 );

 if (_hasCallback(executionData.callbackData)) {
▷ handleAfterPrincipalTransferCallback(_loan, msg.sender,
    executionData.callbackData, totalFee);
 }
```

```
_processRepayments(_loan);
```

- A callback observes mixed identity: old principal, tranches, duration, startTime, protocolFee, and `lenderRefinanceDisabled` together with the new origination fee.
- Currently, `PurchaseBundler` is unaffected: it only reads borrower, principalAddress, nftCollateralAddress, and nftCollateralTokenId, which refinance checks keep equal.

Recommendations:

Pass the newly built `loan` into `handleAfterPrincipalTransferCallback`, matching `emitLoan` and `ILoanCallback`.

For example:

```
// src/lib/loans/MultiSourceLoan.sol
 if (_hasCallback(executionData.callbackData)) {
 handleAfterPrincipalTransferCallback(loan, msg.sender,
    executionData.callbackData, totalFee); //@audit newly built loan, not
    `_loan`
 }
```

Gondi: Resolved with @94f6b264c1 ...

Zenith: Verified.

[I-2] `lenderRefinanceDisabled` blocks borrower-initiated renegotiation and tranche additions, not only lender refinances

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

Target:

- MultiSourceLoan.sol#L693-L699

Description:

The field is named `lenderRefinanceDisabled`, which reads as if it only stops other lenders from refinancing the loan. Setting it disables every in-place renegotiation of the loan, including the ones the borrower initiates for their own benefit.

`_baseRenegotiationChecks` reverts for any protected loan, and it runs at the top of `refinanceFull`, `refinancePartial`, and `addNewTranche` before their caller-specific logic. On a protected loan this means:

- a competing lender cannot refinance the loan (the behavior the name describes)
- the borrower cannot accept a better-terms renegotiation through the borrower-initiated branch of `refinanceFull`
- the borrower cannot add a tranche through `addNewTranche`

A borrower who accepts a protected offer gives up renegotiation flexibility the field name does not advertise. The broad block is required for the invariant. The borrower-initiated branch of `refinanceFull` can route an incoming lender's funds to the old protected lender, and `addNewTranche` would turn the loan multi-tranche, so both have to be blocked to keep the single-tranche, borrower-settled guarantee. The borrower keeps one renegotiation route, `refinanceFromLoanExecutionData` with `principalReceiver == borrower`, which preserves the settlement property. The point is that the field name and any lender- or borrower-facing documentation understate the scope of the opt-out.

Recommendations:

Document the flag as a loan-level opt-out of all in-place renegotiation and tranche additions, and surface that scope to both parties before an offer is signed. If a name change is acceptable, something like `renegotiationDisabled` describes the behavior more accurately.

Gondi: Resolved with PR-628.

Zenith: Verified.

[I-3] The `lenderRefinanceDisabled` transfer guarantee holds at settlement but is not a per-transfer property

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

Target:

- MultiSourceLoan.sol#L354-L359
- LiquidationDistributor.sol#L121

Description:

The invariant that needs to be always true is that a lender who sets `lenderRefinanceDisabled` is only ever paid by the loan's own borrower.

That invariant holds at the settlement boundary. On a protected refinance, `refinanceFromLoanExecutionData` forces `executionData.principalReceiver == loan.borrower`, and `_processRepayments` pays the existing lender with `safeTransferFrom(loan.borrower, tranche.lender, repayment)`. The incoming lender's funds reach the borrower, and the borrower pays the protected lender.

"Every transfer this lender ever receives comes from the borrower," the invariant is wider than the contract can enforce. When a protected loan defaults, a pool lender is paid from the liquidator during the auction, and a non-pool single-tranche lender receives the collateral NFT on foreclosure. Any address can also send ERC20 to a lender at any time, including a marketplace module during the callback that runs before `_processRepayments`. These transfers reach the lender from a source other than the borrower, yet all of them leave the settlement guard intact: the protected lender's repayment is hardcoded to come from `loan.borrower`.

The gap is between the invariant as stated (every transfer comes from the borrower) and the invariant the code actually upholds (settlement pays the lender only from the borrower, default pays from the liquidator or hands over the collateral).

Recommendations:

Consider to state the guarantee at the settlement boundary rather than over all transfers, for example: "direct refinance and repayment settlement must not pay an existing lender from any address other than `loan.borrower`."

Gondi: Resolved with PR-627.

Zenith: Verified.

[I-4] One pool can originate loans on both v3.1 and v3.2

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

Target:

- src/lib/pools/PoolOfferHandler.sol#L138-L151

Description:

`PoolOfferHandler.validateOffer` is version-agnostic: it decodes `OfferExecution`, requires empty `validators`, and checks APR terms. Empty-validator encodings still `abi.decode` across the v3.2 `LoanOffer` layout change, so `addCaller` is the only isolation between MSL versions. Callers cannot be removed.

```
// src/lib/pools/PoolOfferHandler.sol
 function validateOffer(uint256, bytes calldata _offer)
    external view override returns (uint256, uint256) {
▷ IMultiSourceLoan.OfferExecution memory offerExecution
    = abi.decode(_offer, (IMultiSourceLoan.OfferExecution));
 if (offerExecution.offer.validators.length ≠ 0) {
 revert InvalidOfferError();
 }
 uint256 apr = _terms[
  offerExecution.offer.nftCollateralAddress][
  offerExecution.offer.duration][
  offerExecution.offer.maxSeniorRepayment][
  offerExecution.offer.principalAmount];
 if (offerExecution.offer.aprBps < apr || apr == 0) {
 revert InvalidAprError();
 }
 return (offerExecution.amount, offerExecution.offer.aprBps);
 }
```

```
// src/lib/loans/LoanManager.sol
 /// @dev Given repayments, we don't allow callers to be removed.
 function addCaller(ProposedCaller calldata caller) external onlyOwner {
 ...
▷ _acceptedCallers.add(caller.caller);
```

```
_isLoanContract[caller.caller] = caller.isLoanContract;
afterCallerAdded(caller.caller);
}
```

This manifests if a pool that already lists a v3.1 MSL also `addCallers` v3.2. That pool can then originate on both versions. `validateOffer` never reads v3.2-only `LoanOffer` fields, so a shared handler cannot assume v3.2-only policy.

- The same pool liquidity can originate on v3.1 and v3.2.
- Dual listing is permanent: callers cannot be removed after `addCaller`.
- A multi-version pool should keep `validateOffer` generic; a v3.2-only pool can constrain `lenderRefinanceDisabled` and `offer.borrower`.

Recommendations:

(1) Confirm whether each pool is meant to be v3.2-only or to serve multiple MSL versions.

(2) If the pool is v3.2-only: `addCaller` only the v3.2 MSL. `validateOffer` can then enforce this pool's policy on `lenderRefinanceDisabled` and `offer.borrower`.

(3) If the pool is meant to work with multiple MSL versions: keep `validateOffer` generic (empty `validators` and terms only). Existing pools: `addCaller` v3.2 only if that liquidity should still originate on v3.2.

Gondi: Resolved with `@d2e750aae6 ...`

Zenith: Verified.

[I-5] v3.2 MultiSourceLoan ABI cannot share v3.1 companions, and no YAML deploys replacements

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

Target:

- src/interfaces/loans/IMultiSourceLoan.sol#L28-L44

Description:

`LoanOffer` and `Loan` gain trailing fields, so every external function, event, callback selector, and inlined `Hash` that embeds those structs is a new artifact. Live v3.1 liquidator, distributor, PurchaseBundler, and `IOfferValidator` implementations still encode the old tuple and cannot serve v3.2 `MultiSourceLoan`. `deploy/deployments/` has `v3/` and `v3.1/` only — there is no `v3.2/` YAML that deploys the replacements.

`deploy_key` is `Contract.name_str`. Keys left in YAML are skipped, so a v3.2 MSL deploy that still lists those companion keys is constructed against v3.1 bytecode, which will cause on-chain DOS due to ABI conflicts.

```
// src/interfaces/loans/IMultiSourceLoan.sol
 struct LoanOffer {
 ...
 IBaseLoan.OfferValidator[] validators;
▷ bool lenderRefinanceDisabled;
 address borrower;
 }

 struct Loan {
 ...
 uint256 protocolFee;
▷ bool lenderRefinanceDisabled;
 }
```

Companions take that `Loan` (or `LoanOffer`) in their ABI, for example:

```
// src/interfaces/ILoanLiquidator.sol
 function liquidateLoan(
```

```
 uint256 _loanId,
▷ IMultiSourceLoan.Loan calldata _loan,
 uint96 _duration,
 uint256 _minBid,
 address _originator
 ) external returns (bytes memory);
```

The same layout is in `LiquidationDistributor.distribute`, `ILoanCallback.afterPrincipalTransfer` / `afterNFTTransfer` (PurchaseBundler), and `IOfferValidator.validateOffer`. v3.1 addresses still fill the only mainnet recipe:

```
# deploy/deployments/v3.1/mainnet.yml
 AuctionWithBuyoutLoanLiquidator:
    '0x2995AE7233fA89b314b5a707465B57a582F440F0'
 ...
 PurchaseBundler: '0xfd31a0cd628f0bab2cc174c3abd6bfc2d01aca61'
 RangeValidator: '0x039BC1010f0295246d8004224600D65d804F2b0A'
 MultiAddressValidator: '0xDDCE55Af28FCD6C3F5C9A35D5a0aDa9c8f103aa0'
```

Trigger: ship this commit using a copy that omits only `MultiSourceLoan`.

Impact:

- v3.2 `liquidateLoan` / `settleAuction` / BNPL callbacks / validator calls miss v3.1 selectors or fail `abi.decode` (fail-closed mixed cluster).
- Shared managers (`CURRENCY_MANAGER`, `COLLECTION_MANAGER`, `LoanManagerRegistry`) stay compatible; existing v3.1 loans on the old MSL are unaffected.

Recommendations:

Add `deploy/deployments/v3.2/mainnet.yml` that omits ABI-sensitive keys so those contracts redeploy against the new `Loan` / `LoanOffer` layout, and keep the shared managers.

Gondi: Resolved with `@024feb6c30 ...`

Zenith: Verified.

[I-6] Unused LoanOffer getInterest method

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

Target:

- src/lib/utils/Interest.sol#L6-L17

Description:

`Interest.getInterest(IMultiSourceLoan.LoanOffer memory)` is never called from `src/`. Fill, pool, and liquidation paths use `using Interest for uint256` and the scalar overload.

```
// src/lib/utils/Interest.sol
import "@solmate/utils/FixedPointMathLib.sol";
import "../../interfaces/loans/IMultiSourceLoan.sol";
▷ import "../../interfaces/loans/IBaseLoan.sol";

library Interest {
 using FixedPointMathLib for uint256;
 ...
▷ function getInterest(IMultiSourceLoan.LoanOffer memory _loanOffer)
    internal pure returns (uint256) {
 return _getInterest(_loanOffer.principalAmount, _loanOffer.aprBps,
    _loanOffer.duration);
 }

 function getInterest(uint256 _amount, uint256 _aprBps, uint256 _duration)
    internal pure returns (uint256) {
 return _getInterest(_amount, _aprBps, _duration);
 }
```

Recommendations:

Delete `getInterest(LoanOffer memory)`. Keep the library, the scalar `getInterest`.

Gondi: Resolved with `@f37d4a0e85 ...`

Zenith: Verified.

[I-7] Integrations routing principal to a borrower-controlled address break on flagged refinance

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

Target:

- src/lib/loans/MultiSourceLoan.sol#L354-L359

Description:

`refinanceFromLoanExecutionData` on a loan with `lenderRefinanceDisabled` reverts unless incoming principal is credited to `_loan.borrower`, breaking integrations that always route principal to a borrower-controlled vault or bundler. `emitLoan` does not apply that gate, so the same pattern succeeds at origination.

```
// src/lib/loans/MultiSourceLoan.sol
 /// @dev A protected lender is repaid by the borrower out of
    `_processRepayments`. Letting the
 /// borrower point the incoming principal elsewhere would route the new
    lender's funds
 /// straight to them instead, which is what the flag exists to prevent.
▷ if (_loan.lenderRefinanceDisabled && executionData.principalReceiver ≠
    borrower) {
 revert LenderRefinanceDisabledError();
 }
```

`emitLoan` passes the same field through with no equivalent check:

```
// src/lib/loans/MultiSourceLoan.sol
 (uint256 loanId, uint256[] memory offerIds, Loan memory loan,
    uint256 totalFee) = _processOffersFromExecutionData(
 borrower,
 executionData.principalReceiver,
 ...
 );
```

The fill path credits `_principalReceiver`; `_processRepayments` still pulls from

`loan.borrower`:

```
// src/lib/loans/MultiSourceLoan.sol
 ERC20(offer.principalAddress).safeTransferFrom(lender, _principalReceiver,
    amount - fee);
```

```
// src/lib/loans/MultiSourceLoan.sol
 asset.safeTransferFrom(loan.borrower, tranche.lender, repayment);
```

The live loan has `lenderRefinanceDisabled`. The caller sets `executionData.principalReceiver` to a borrower-owned vault, bundler, or router (`V` $\neq$ `borrower` and `V` $\neq$ `tranche[0].lender`), including PurchaseBundler `swapAndExecute` targeting this selector.

- Shared `ExecutionData` builders that always set `principalReceiver` to an owned vault, bundler, or router succeed at flagged `emitLoan` and revert at refinance-from-offers. This could break existing integrations.
- No fund loss or frozen designed exit: `repayLoan` and refinance with `principalReceiver` $=$ `borrower` still work.

Recommendations:

(1) If the invariant is payer-side only, revert when `principalReceiver` equals the outgoing flagged lender (`_loan.tranche[0].lender`), not when it differs from `_loan.borrower`.

For example:

```
// src/lib/loans/MultiSourceLoan.sol
 if (
 _loan.lenderRefinanceDisabled &&
 executionData.principalReceiver == _loan.tranche[0].lender
 ) { //@audit revert only if new principal would be paid to the protected
    lender
 revert LenderRefinanceDisabledError();
 }
```

(2) If new principal must land on `loan.borrower` even when that is stricter than the payer invariant, keep the current check and document that protected origination and protected refinance do not share a receiver policy, and existing integration for a protected lender refinance needs to strictly set principalReceiver as the borrower.

Gondi: Resolved with `@f13a32ee76 ...`

Zenith: Verified.

[I-8] The two new features together give a lender a targeted, uncontested foreclosure-to-own right on a specific NFT

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

Target:

- MultiSourceLoan.sol#L474
- LiquidationHandler.sol#L100-L105
- MultiSourceLoan.sol#L697-L699
- MultiSourceLoan.sol#L357-L359

Description:

On default, `liquidateLoan` forecloses, it transfers the collateral straight to the sole lender with no auction, when `tranche.length == 1 && !isLoanManager(tranche[0].lender)`:

```
(bool foreclosed, bytes memory liquidation) = _liquidateLoan(
 _loanId, _loan, _loan.tranche.length == 1 &&
    !getLoanManagerRegistry.isLoanManager(_loan.tranche[0].lender)
);
```

`lenderRefinanceDisabled` pins the loan to a single tranche and blocks every path that could add a tranche or replace the lender: `addNewTranche`, `refinancePartial`, and both branches of `refinanceFull` all revert in `_baseRenegotiationChecks`. For a non-pool lender this makes the default outcome deterministic, they receive that exact NFT, with no competing bidders. The `borrower` pin restricts the offer to one named borrower, so a lender can target a specific borrower who holds a specific NFT.

Before these new features, a lender could not guarantee this. The borrower could add a defensive tranche, which makes `tranche.length` two and sends the loan to auction instead of foreclosure, or a competing lender could refinance the loan away. `lenderRefinanceDisabled` removes both routes.

The borrower keeps two exits: full repayment, and self-refinance through `refinanceFromLoanExecutionData` with `principalReceiver == borrower`. The second works only if a third lender will fund a new loan large enough to repay the protected one. A lender who sets the flag at high LTV on a short duration removes that option in practice, leaving the borrower with repay-in-cash or lose the NFT. Auction surplus offers no

protection either: proceeds above the debt go to the lenders in both the foreclosure and auction paths, so the borrower forfeits equity regardless of path.

The payoff of such a loan is principal plus interest if repaid, otherwise this specific NFT, with the borrower's in-protocol counter-moves disabled. Both new fields sit inside the offer hash the borrower signs, so the borrower consents to the terms. The field name `lenderRefinanceDisabled` does not convey any of this to the borrower.

Recommendations:

Decide at the product level whether a loan-to-own line is intended. If it is, disclose it to borrowers at signing: a protected offer waives passive rate improvement, tranche additions, and the auction backstop, and lets the lender take the collateral uncontested on default. If it is not, consider disallowing the flag above an LTV threshold.

Gondi: Resolved with PR-632.

Zenith: Verified.