Quantstamp

Gondi

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	NFT Lending Protocol
Timeline	2023-10-04 through 2023-10-20
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Unpublished internal documentation about Gondi V2
Source Code	• https://github.com/pixeldaogg/florida-contracts #b617e7e
Auditors	• Ruben Koch Auditing Engineer • Ibrahim Abouzied Auditing Engineer • Julio Aguliar Auditing Engineer

Documentation quality	Medium
Test quality	Medium
Total Findings	19 Fixed: 13 Acknowledged: 5 Mitigated: 1
High severity findings	3 Fixed: 3
Medium severity findings	0
Low severity findings	12 Fixed: 8 Acknowledged: 3 Mitigated: 1
Undetermined severity findings	0
Informational findings	4 Fixed: 2 Acknowledged: 2

Summary of Findings

In this audit, we reviewed the Gondi V2 NFT lending platform. Borrowers and Lenders can offer loan terms that the other party can agree to. The borrower's NFT (or bundle of NFTs/tokens if a specific vault is leveraged) is put in escrow in exchange for the lender's principal. A loan can be repaid at any time, provided all accrued interest rate and principal is returned. If a borrower fails to do so, the NFT is either claimed by the single lender of the loan or auctioned off, with the proceeds of the auction being distributed to the partial lenders.

Loans can be partially or fully refinanced by any lender, provided that the specified APR will result in sufficiently lower daily interest to be paid by the borrower. A loan can also be renegotiated, where a borrower accepts a loan offer that is not necessarily strictly better but offers a desired adjustment, e.g. extended loan duration for increased APR. Refinanced sources continue to accrue a small APR, as a portion of the delta in APR of the refinancing source continues to accrue interest for refinanced lenders. These "proceeds" are however discarded in case of liquidations.

Loan repayment and loan initialization can be paired with hooks, where certain actions involving the loan's principal or collateral can be performed. This enables Buy Now Pay Later functionality, where the borrower essentially performs a leveraged buy of the collateral with the loan's principal and also enables the market selling of the collateral NFT to repay a loan.

In the course of the audit, a few significant concerns arose. The `UserVault` can be drained from all ERC721 tokens (GON-1) and the callback data from borrower-signed protocol interactions remains unsigned, enabling anyone to manipulate it, potentially removing the full difference between e.g. the sale price and the loan costs from the borrower (GON-2). The proceeds mechanic can also be abused to force loans into liquidation, as the array can grow indefinitely, making the settlement of proceeds debt impossible, and forcing the loan to get liquidated (GON-3). In multiple instances, we also identified a lack of input validation (GON-5, GON-14).

While we deem the code quality to be high, the codebase could benefit from added code comments for improved readability. Additional documentation containing a specification of e.g. the flow of funds in the different protocol interactions.

Many tests were provided. However, the test suite currently does not support any coverage metrics and did not execute with other internal tooling, making it difficult to judge the test suite's thoroughness.

Update Fix-Review The proceeds mechanic has been removed from the protocol, as a result, certain issues no longer apply. These issues have been marked as "Fixed". All of the remaining issues have either been acknowledged or properly fixed by the client.

ID	DESCRIPTION	SEVERITY	STATUS
GON-1	Missing Ownership Check In Vault	● High	Fixed
GON-2	Unsigned Callback Data	● High	Fixed
GON-3	Proceeds Array Can Be Arbitrarily Extended	● High	Fixed
GON-4	Missing Existence Check in Vault	● Low	Fixed
GON-5	Borrower Lacks Capability to Cancel Requests	● Low	Fixed
GON-6	Privileged Roles and Ownership	● Low	Acknowledged
GON-7	Funds Could Get Stuck In Leverage Contract	● Low	Fixed
GON-8	Lenders Can Claim NFT With Last-Second Full Loan Refinance	● Low	Acknowledged
GON-9	Loan Requests with Unspecified Lender Can Behave Unexpectedly	● Low	Fixed
GON-10	Possible Reentrancy When Creating Loans	● Low	Fixed
GON-11	Unexpectedly High Tax for Borrower	● Low	Fixed
GON-12	Greedy `UserVault` Contract	● Low	Fixed
GON-13	Unpaid Protocol Proceeds Fee in Loan Renegotiation	● Low	Fixed
GON-14	Missing Input Validation	● Low	Mitigated
GON-15	Critical Role Transfer Not Following Two-Step Pattern	● Low	Acknowledged
GON-16	Loans Have Overly Constrained Capacity	● Informational	Fixed
GON-17	Liquidation Process Can Be Denied	● Informational	Fixed
GON-18	Incompatibility with Non-Standard Tokens	● Informational	Acknowledged
GON-19	Unlocked Pragma	● Informational	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer
Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors

- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
   1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

In scope of the audit where all parts of Gondi V2 release contained in the `src/` folder, the only notable exception being all aspects around the validators. The validator contracts can be employed for so-called collection offers, where lender or borrower leaves the specific `tokenId` of a loan offer unspecified, only specifying a set of validators that need to approve a `tokenId` added by either the accepting lender or borrower. The aspects of this mechanic was not part of the review.

Files Included

Repo: https://github.com/pixeldaogg/florida-contracts(b617e7e7339ad5c23ab396d873203bfe6870aa2d) Files: src/

Files Excluded

Repo: https://github.com/pixeldaogg/florida-contracts(b617e7e7339ad5c23ab396d873203bfe6870aa2d) Files: src/lib/validators and src/lib/utils/ValidatorHelpers.sol

Findings

GON-1 Missing Ownership Check In Vault

● High ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `37fd1a6c3040d1a7f685e487715c0c9de9be827a`.

File(s) affected: `UserVault.sol`

Description: Users can bundle up NFT and tokens in a special NFT called vaults via the `UserVault` contract. Tokens and NFTs bundled in such a vault are only accessible once the vault is burned again. The idea is that these vaults containing a set of NFTs and tokens can be used as collateral, as they give the owner access to the underlying assets once it is burned. The contract provides multiple functions to withdraw the different assets. While `withdrawTokens()` and `_withdrawEth()` work as expected, the `_withdrawNfts()` function does not check whether the vault held possession of the to-be-withdrawn NFT. As the minting and burning of vaults are done permissionlessly, anyone can drain all NFTs the contract holds.

Exploit Scenario:

1. A malicious user creates a vault via `UserVault.mint()`, receiving a `_vaultId`
2. In another transaction, the user calls `burnAndWithdraw()`, specifying the newly minted `_vaultId` and all NFTs they wish to drain. The `tokens` array must be kept empty, as it has proper checks in place that would cause the transaction to revert if the user would specify tokens to withdraw they did not previously deposited.
3. In the `_withdrawNft()` function, the `onlyReadyForWithdrawal` modifier would pass, as the user is registered in the `_readyForWithdrawal` mapping for the `_vaultId`.
4. The NFT is transferred and the entry in the `_vaultNfts[_collection][_tokenId]` mapping is deleted, without assuring that its previous value matched the specified `_vaultId`. That way, all NFTs are accessible through any vault.

Recommendation: Assure that `_vaultNfts[_collection][_tokenId] == _vaultId` before allowing the transfer in `_withdrawNft()`. Also, follow the Check-Effects-Interacts-pattern for best practice.

GON-2 Unsigned Callback Data

• High ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `a05d5e8223351b0a3466bd1bcde507493fde0f24`.

The callback data in all cases is now always required to be signed by the borrower if the borrower is not `msg.sender`.

File(s) affected: `Leverage.sol`, `MultiSourceLoan.sol`

Description: The protocol enables the initialization and repayment of loans paired with callback data, which enables hooks to whitelisted contracts with provided calldata. With that, users can take a loan with the principal of the to-be-taken loan being used to buy the collateral in the first place ("buy now pay later"/"BNPL"). It can also be used to pair the selling of the NFT directly with its repayment.

For such a request, the borrower does not necessarily need to be the `msg.sender`, but can also sign parts of the loan information instead, which is then verified on-chain. However, the callback data for the hooks remains unsigned, which is very problematic, as front-runners can replace the callback data. This is problematic in both cases of the hook usage:

For loan repayments via selling `_repaymentData.callbackData` can be arbitrarily adjusted to possibly perform market sells of the NFT to repay the loan that return just enough funds to the borrower to cover the loan expenses. While the contract being interacted with still has to be whitelisted, any data regarding which function to invoke with what values and how much (w)ETH to transfer can be manipulated. While this is still a fairly constrained environment, where a certain contract has to be invoked and enough funds have to be made accessible to the borrower to repay the loan (assuming they currently do not have any available), it can leave the borrower at a significant loss.

For example, assume a borrower took a 1 ETH loan with 1% APR against a 50 ETH floor price NFT and wants to repay the loan after a year by selling the NFT and get the rest of the profit. If they sign as part of the `LoanRepaymentData`, anyone can replace the `callbackData` in that struck to edit in a sale to an offer as low as 1.1 ETH to settle all necessary debt, leaving the borrower at a massive loss.

- For BNPL interactions, where a loan is taken and from the lended principal the collateral is purchased via the `_handleAfterPrincipalTransferCallback()`, the `callbackData` too could be replaced in a way that could be unintended for the borrower. While the function seems to be designed for leveraged buys, where the loan is fully used with added funds from the borrower (see GON-7), it is still an unnecessary risk to leave the `callbackData` unsigned.

While it should be noted that such an attack seems not possible for Seaport offer filling, as both parties need to have signed and thereby agreed upon an offer beforehand, but Seaport is just one of the planned integrations and the code is already present to seamlessly integrate more that possibly have looser constraints.

Recommendation: Non-zero `LoanExecutionData.callbackData` should be signed by the borrower, as well as non-zero `LoanRepaymentData.callbackData` in case of repayments.

GON-3 Proceeds Array Can Be Arbitrarily Extended

• High ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `f4279fef22a11d9b288dc378b2dd764c0a3fd166`.

The proceeds mechanic has been fully removed in the specified commit. Therefore, the issue is no longer relevant.

File(s) affected: `MultiSourceLoan.sol`

Description: In partial and full refinances, refinanced sources continue to receive a portion of the difference between the APR they originally offered and the APR the currently refinancing source offers. This concept is called proceeds and theoretically continues to accrue interest until a loan is repaid or renegotiated. This incentivizes early lenders to accept loans, as they might receive a passive income stream in return even if they get immediately refinanced.

These proceeds are tracked in a separate dynamically sized array as part of the `Loan` struct. It is traversed and possibly appended to in nearly all interactions with a loan. It can become fully discarded only via renegotiations, where all proceeds are paid off by the borrower. In case of liquidations, the proceeds are not considered.

The proceeds are only paid out in case of renegotiations and loan repayment. The array is appended to for each source that is being refinanced, so `_loan.sources.length` elements per transaction. It should be noted that the maximum number of sources a loan can be partitioned in is capped by `MultiSourceLoan._maxSources`, which according to the documentation will be set to ten. While in most cases, only a full refinance would possibly extend the array by e.g. ten, leaving follow-up refinances to only extend the array by one, a malicious user could possibly fully refinance all sources not via `refinanceFull()` but via `refinancePartially()`, refinancing each of the e.g. all ten sources individually, creating ten new entries in the `proceeds` array, while the loan continues to have ten active sources. With each such refinance, the proceeds array would grow by ten elements.

If a user attempts to repay or renegotiate such a loan, they would be forced to settle the interest accumulated by all proceeds. This would require an ERC20 transfer for each of the refinanced sources throughout a loan's lifetime (or until its last renegotiation). The gas costs of an ERC20 `transferFrom()` call differ by implementation, but can be approximated by 65000 gas. With a block gas limit of 30 million on mainnet, there is a rough theoretical capacity of 461 transfers that can be handled within one transaction, though the reality will be much lower than that, given all the associated gas costs of e.g. the `repayLoan()` function.

While significant gas costs would be associated with performing such a bloating attack, it would force a borrower into liquidation.

Recommendation: Switch to proceeds, where users can withdraw funds tracked in some `totalProceeds` mapping that gets increased any time the funds would be pushed to the original lender.

GON-4 Missing Existence Check in Vault

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `27e62624bd9ffae5c4f4a6c85b2ba2d2a5a3e366` and `753c5272f216e2944adac852517c1bc5d84a3a71`.

A `vaultExists` modifier has been added to perform an existence check prior to deposits.

File(s) affected: `UserVault.sol`

Description: None of the deposit functions validate that the provided vault exists. The NatSpec comment above `depositNft()` states that the Gondi team does not check that the vault exists and it is the responsibility of the caller to make sure of that.

However, the DeFi user experience is still very poor and is therefore error-prone, especially for the vast majority of (new) users. Not validating that the vault exists could lead to users being unable to withdraw their NFTs, ERC20s, and ETH since they would have to first burn the non-existent vault ID which would revert.

Recommendation: Validate that the vault exists prior to any deposits via a comparison with `totalSupply`.

GON-5 Borrower Lacks Capability to Cancel Requests

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `7916710f87139597202127455a5e18a0ed844f1c`.

The borrower can now also cancel requests.

File(s) affected: `MultiSourceLoan.sol`, `BaseLoan.sol`

Description: The `LoanOffer` struct used to publish a loan on-chain supports multiple flows where finally, both borrower and lender are required to commit to the `LoanOffer` for a loan to be initiated, either via a signature or by being the sender of the transaction.

A borrower can sign the `ExecutionData` containing a loan they would like to request and either provide a specific lender able to carry out the loan request or leave that field unset to enable anyone to lend the principal. Once they have signed such an `ExecutionData`, it is impossible for the requesting borrower to cancel the request. The only way a borrower can prevent the loan request from being filled is by revoking the approval of the specified NFT or by waiting for the loan request to expire.

This is asymmetrical to the capabilities of a lender, who can cancel certain loan offers of a certain `offerId` at any time via `cancelOffer()` or `cancelRenegotiationOffer()`.

Recommendation: Provide the capability to the borrower to cancel loan requests too and add the checks for it as part of the `_validateExecutionData()` function call.

GON-6 Privileged Roles and Ownership

• Low ⓘ Acknowledged

Update
The client marked the issue as "Acknowledged".

Description: Certain contracts have state variables, e.g. `owner`, which provide certain addresses with privileged roles. Such roles may pose a risk to end-users.

For a detailed list of roles per contract see the following list:

The owner of the `MultiSourceLoan` contract has access to:

1. `setDelegateRegistry()` to update the delegate registry
2. `setMaxSources()` to update the total number of sources a loan's principal can be partitioned in.
3. `updateRefinanceInterestFraction()` to update the percentual share of how much of the delta of a refinanced source the original lender is entitled to in case of loan repayment and renegotiations.
4. `setFlashActionContract()` to update the address that a borrower can execute flash loans with when their asset is locked as collateral.

5. Via inheritance of the abstract `WithCallbacks` contract, the `MultiSourceLoan` contract furthermore has access to:
   a) `addWhitelistedCallbackContract()` and `removeWhitelistedCallbackContract()` to maintain the list of whitelisted callback contract instances that can reenter after the execution of callback hooks.
6. Via inheritance of the abstract `BaseLoan` contract, the `MultiSourceLoan` contract furthermore has access to:
   b) `updateProtocolFee()` and `setProtocolFee()`, offering a two-step update mechanism to overwrite the protocol fee deducted at various places in the protocol.
   c) `updateLiquidationContract()` to update the contract to use for liquidations. Overwriting this contract with incorrect values could cause the loss of all collateral of defaulted loans.
   d) `updateLiquidationAuctionDuration()` to update the base duration of the auction outside of the quiet-ending mechanic.

The owner of the `AuctionLoanLiquidator` contract has access to:

1. `addLoanContract()` and `removeLoanContract()` that maintains a list of approved loan contracts that this contract can perform liquidations for.
2. `updateTriggerFee()`, which enables the liquidator to specify the fees transferred to the initiators and finalizers of the liquidation process, with an upper bound of 0.5% for both.

The owner of the `AddressManager` contract has access to:

1. `add()`, `addtoWhiteList()`, `removeFromWhitelist()`, which maintains a list of whitelisted addresses. It is leveraged to maintain a whitelist of ERC20 tokens approved for principal/payments and a whitelist of ERC721 token addresses approved as collateral. The `Leverage` contract also maintains a list of the approved external marketplaces that can be used in loans emitted together with hooks.

The owner of the `Leverage` contract has access to:
1. `updateMultiSourceLoanAddressFirst()` and `finalUpdateMultiSourceLoanAddress()`, offering a two-step update mechanism to overwrite the address able to access the hook functions.
2. `updateSeaportAddressFirst()` and `finalUpdateSeaportAddress()`, offering a two-step update mechanism to update the `_seaport` address that can be used in external hooks.

Recommendation: These privileges should be made clear to the users via documentation.

GON-7 Funds Could Get Stuck In Leverage Contract

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `753c5272f216e2944adac852517c1bc5d84a3a71`.

File(s) affected: `Leverage.sol`

Description: The `Leverage` contract can be used for protocol interactions paired with callbacks to execute once principal from a loan is received (`Leverage.afterPrincipalTransfer()`) or the loan's underlying collateral is received (`Leverage.afterNFTTransfer()`).

However, in both of these callbacks, funds could get permanently stuck in the contract, as the `afterPrincipalTransfer()` callback assumes that the combination of `msg.value` and the loan's principal will exactly cover the purchase attempted in the callback. In `afterNFTTransfer()` the returned amount is also specified in the callback data and might not be fully aligned with the received funds.

Recommendation: As the contract is not intended to hold any funds, we believe the contract should simply return the contract balance, if the contract balance is greater than any specified amount.

GON-8 Lenders Can Claim NFT With Last-Second Full Loan Refinance

• Low ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> "We thought already about this and it's fine. All solutions are worse than existing state given they require defining a time window."

File(s) affected: `MultiSourceLoan.sol`

Description: Loans can be refinanced at any moment by any lender. If a borrower defaults on the loan, the NFT would get automatically transferred to the lender (if there is a single lender). Since there is also no constraint to refinancing besides making the loan better for the borrower, lenders might front-run each other for full refinance requests at the end of the loan duration to get the NFT. As it is fairly unlikely that a loan will be paid back in one of the last blocks, there is no downside for lenders to engage in such behaviour, if the NFT is worth more than the principal.

Given that the loan would have a very small duration left, a refinance with e.g. the minimum possible APR of 0.01% of would not leave the new lender's funds ineffectively idle for long, yet make follow-up refinances impossible, locking the NFT claim to the lender in case of loan default.

Recommendation: Provide end-user-facing documentation detailing this fact. Alternatively, consider mitigating this by not allowing refinances without significantly extended loan duration once the remaining duration of the loan has gone below some threshold. As some duration would be left, the borrower might still have the opportunity to repay the loan to reclaim the NFT themselves, increasing the risk of idle funds for last second full refinances before the threshold is reached.

GON-9 Loan Requests with Unspecified Lender Can Behave Unexpectedly

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `21ad657dd4bfed780fdca288bc4ea3af444167dd`.

File(s) affected: `MultiSourceLoan.sol`, `BaseLoan.sol`

Description: A borrower can request a loan that any lender can take by specifying a `LoanOffer` with an unspecified `lender` field.

If for some reason, a borrower would ever sign a collection loan offer request with an unspecified lender, yet `capacity > 0`, there is the potential for unexpected behavior if some conditions are met. In general, all loan requests with unspecified lenders about to be put on-chain via `emitLoan()` can be front-ran by other lenders to overtake the loan. The only requirement for such cases would be that the front-running lender replaces their address in the `LoanExecutionData.lender` field.

There are more specific cases, depending on the `offer.capacity` field.

In cases of `offer.capacity > 0`:

- Other borrowers can mark the capacity of that loan as used under loan conditions they fully determine themselves, as long as the lender in the `LoanOffer` also remains unspecified and the `offerId` matches, since `_used[offer.lender][offer.offerId]` would be shared between the two separate loans. However, `capacity > 0` for borrower-signed loan requests seems rather unuseful, as the borrower is required to sign off a specific NFT to use, which could only be repeatedly used if a loan is repaid before the underlying loan offer expires.

In case `offer.capacity == 0`:

- Anyone can deny such a loan request by front-running the emitting of the loan with an arbitrary loan that also leaves the lender unspecified and matches the `offerId` of the other loan. As the two loans share the unspecified lender and `offerId` can be arbitrarily forged, it is marked as canceled in the `isOfferCancelled` mapping.

Recommendation: In case `offer.lender` remained unspecified, consider using the borrower in the first index of the `used` mapping.

GON-10 Possible Reentrancy When Creating Loans

• Low ⓘ Fixed

Update
The `emitLoanMany()` function has been removed in commit `cce2e1dc7468b5eb23ea4d0c7c57eca093eecde2`, so the issue no longer applies.

File(s) affected: `MultiSourceLoan.sol`

Description: The `emitLoan()` function is guarded by the `nonReentrant` modifier to protect against reentrancy. However, the `emitLoanMany()` function is missing the modifier, allowing users to circumvent the protection.

Recommendation: Though no exploit was identified, add the `nonReentrant` modifier to the `emitLoanMany()` function to further protect the contract.

GON-11 Unexpectedly High Tax for Borrower

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `abd99ad352b9f397f003eea31693daf024fccf79`.

File(s) affected: `MultiSourceLoan.sol`

Description: In case borrowers draw from a line of credit with some maximum capacity, the possible `taxCost` the borrower has to pay to the lender should not be based on the loan's maximum size, i.e. `offer.principalAmount`, but instead be based on the value the borrower is borrowing, which is the value of the `amount` variable.

Recommendation: Base the `taxCost` to be paid off of the `amount` value, not the `offer.principalAmount` value.

GON-12 Greedy `UserVault` Contract

• Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `27e62624bd9ffae5c4f4a6c85b2ba2d2a5a3e366`.

File(s) affected: `UserVault.sol`

Description: The `UserVault` contract provides both a `fallback()` and `receive()` function. All native tokens deposited via direct transfers would be forever stuck in the contract, as only funds received via `depositEth()` would ever be withdrawable.

Recommendation: Remove both the `fallback()` and `receive()` function from the contract.

GON-13 Unpaid Protocol Proceeds Fee in Loan Renegotiation

• Low ⓘ Fixed

Update
The proceeds mechanic got removed in commit `f4279fef22a11d9b288dc378b2dd764c0a3fd166`, so the issue no longer applies.

File(s) affected: `MultiSourceLoan.sol`

Description: The function `MultiSourceLoan._clearProceeds()` calculates a fee on top of the accumulated proceeds interest the borrower has to pay in addition to the accrued proceeds interest. This value is returned to the calling function, which is then expected to take care of the fee transfer.

While in loan repayments, the proceeds fee is transferred from the borrower to the fee recipient, the protocol fails to claim the proceeds fee in case of loan renegotiations. The value of the unsent fees from the proceeds are passed regardless into the emission of the `ProceedsCleared()` event.

Recommendation: Add a transfer of `totalProtocolFee` to the fee recepient in case of loan renegotiations.

GON-14 Missing Input Validation

• Low ⓘ Mitigated

Update
Some of the recommendations have been implemented in commit `90daa9287068fbfe5768b3281be06ee7294a25ee`.

Related Issue(s): SWC-123

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error. Specifically, in the following functions arguments could could benefit from additional input validation:

1. Generally, in constructors, address parameters, especially if used to assign immutable variables, should be checked to be unequal to zero. Furthermore, all assignments of global state variable that have matching setters should make use of those setters to leverage the existing input validation.
2. `Leverage.updateMultiSourceLoanAddressFirst()` : Validate that `_newAddress` is non-zero.
3. `Leverage.updateSeaportAddressFirst()` : Validate that `_newAddress` is non-zero.
4. `BaseLoan.updateImprovementMinimum()` : Validate that `_newMinimum` is non-zero to not enable refinances with identical APR.
5. `BaseLoan.updateLiquidationAuctionDuration()` : The function allows the auction duration to be `zero`. Consider adding a minimum check.
6. `BaseLoan.updateImprovementMinimum()` : Validate its input against some minimum values. According to the documentation, these should be `1%` principal amount increase, `1%` interest decrease, or `10%` duration increase.
7. `MultiSourceLoan.setMaxSources()` : Validate that `maxSources` is non-zero and have a reasonable hardcoded upper bound to not cause block gas limit concerns in their processing.
8. `MultiSourceLoan.updateRefinanceInterestFraction()` : Validate that `_newFraction` is lower than `PRECISION.`
9. `AuctionLoanLiquidator.placeBid()` : Validate that `_bid` is greater than zero, as else a bid of zero amount could technically win the auction.
10. `WithCallbacks.addWhitelistedCallbackContract()` : Validate that `_contract` is non-zero and `_tax` is lower than `PRECISION`.
11. `AddressManager._add()` : Validate that `entry` is valid. Otherwise, it would allow `address(0)` to be added to the registry.

Recommendation: We recommend adding the relevant checks.

GON-15 Critical Role Transfer Not Following Two-Step Pattern

• Low ⓘ Acknowledged

Update
The client acknowledged the issue.

File(s) affected: `Leverage.sol` , `BaseLoan.sol` , `AddressManager.sol` , `AuctionLoanLiquidator.sol`

Description: `Solmate/Owned` allows the owner of the inheriting contracts to call `transferOwnership()` to transfer the ownership to a new address. If an uncontrollable address is accidentally provided as the new owner address then the contract will no longer have an active owner, and functions with the `onlyOwner` modifier can no longer be executed.

Recommendation: Consider using OpenZeppelin's `Ownable2Step` contract to adopt a two-step ownership pattern in which the new owner must accept their position before the transfer is complete. Alternatively, add a mechanism similar to e.g. the updating of the variable containing the address of the `MultiSourceLoan` contract in the `Leverage` contract, where both `updateMultiSourceLoanAddressFirst()` and `finalUpdateMultiSourceLoanAddress()` have to be called separately with the correct parameters for the variable to be fully updated.

GON-16 Loans Have Overly Constrained Capacity

● Informational ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `c858936fe217a562d396483b14f1876b9fb3b69f` .

File(s) affected: `BaseLoan.sol`

Description: A lender can propose a loan offer on a collection, allowing for multiple loans to be withdrawn up to a designated `capacity` . `_validateExecutionData()` validates that a new loan has not exceeded the lender's capacity.

However, it checks for whether withdrawing the full `offer.principalAmount` would exceed the capacity, rather than the `executionData.amount` (the actual value that the borrower will be lent). This would prevent the execution of the loan, despite the possibility of the `executionData.amount` remaining within the lender's capacity.

Recommendation: Update the validation for the loan offer's capacity to use `executionData.amount` rather than `offer.principalAmount` .

GON-17 Liquidation Process Can Be Denied

● Informational ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `90daa9287068fbfe5768b3281be06ee7294a25ee` . The client provided the following explanation:

> "Changed to `transferFrom()` "

File(s) affected: `AuctionLoanLiquidator.sol`

Description: There can be a DoS if the winner of the auction of an NFT collateral from a liquidation of a partial loan reverts the `onERC721Received()` callback, as it is distributed via the `safeTransferFrom()` as part of the liquidation process.

A successful bidder of an NFT collateral resulting from the liquidation of a loan with multiple principal sources can deny the payout of the tokens by manipulating the `onERC721Received()` callback, as a transfer of the collateral to the bidder via `safeTransferFrom()` is attempted as part of the liquidation process.

This would result in both the bid and the NFT being permanently frozen, so there is little incentive for a bidder to perform such an attack, but it would still stop lenders from receiving any liquidation proceeds.

Recommendation: Switch to a pull mechanism for the withdrawal of the NFT for the winning bidder.

GON-18 Incompatibility with Non-Standard Tokens

● Informational ⓘ Acknowledged

Update

The client acknowledged the issue with the following explanation:

> "Not a problem."

Description: In general, the protocol has minor limits on what the token can be supported to interact:

- The protocol does not support token charging fees on transfers: Such tokens could cause all protocol interactions to revert, as the accounting relies on the complete flow of funds specified in transfers.
- The protocol does not support rebaseable tokens: the `UserVault` and the `AuctionLoanLiquidator` contract maintain the accounting of token balances in state variables. Tokens with elastic supply (also known as rebasing tokens) and tokens with dishonest implementation would not be compatible with the logic intended by this contract.

- Even though reentrancy guards are used within the protocol, we would strongly discourage the usage of any ERC777 token or any other ERC20 with hooks on transfers.

Recommendation: It is imperative that the team understands the implementation of the ERC20 tokens used within this protocol, and attests to their compatibility before whitelisting the tokens in the protocol.

GON-19 Unlocked Pragma

● Informational ⓘ Acknowledged

Update
The client acknowledged the issue.

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend to remove the caret to lock the file onto a specific Solidity version.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Adherence to Best Practices

2. Fixed In `MultiSourceLoan.sol#L212`, setting the `newSources` array to the fixed length of `1` instead of `totalNewSources` (which is also guaranteed to be `1`) might be more gas efficient.
3. Fixed In `MultiSourceLoan.extendLoan()`, the `_loan.hash() != _loans[_loanId]` check is redundant with the call to `_baseLoanChecks(_loanId, _loan);`.
4. Fixed Make sure to remove unused, commented, or dead code to improve the codebase readability. The following is a non-exhaustive list of such cases:
    2. `Leverage` imports `IBaseLoan` but does not use it.
    3. `Leverage` does not use the errors `InvalidLengthError()`, `NotEnoughEthReceivedError()`, and `InvalidSellError()`.
    4. `BaseLoan` imports `ERC165` but does not use it.
    5. The errors `UnauthorizedError()`, `LoanNotFoundError()`, `ExpiredLoanError()`, and `RepaymentError()` defined in `BaseLoan` are not used by it nor by the `MultiSourceLoan` contract.
    6. `MultiSourceLoan` is not using the errors `InvalidDelegationRegistryError()` and `InvalidFlashActionContractError()`.
5. Fixed It is also recommended to follow naming conventions since it allows for easier code maintenance and better readability among other things. The function `RangeValidator.validateOffer()` declares the variables `min_value` and `max_value` instead of `minValue` and `maxValue`.
6. Fixed Try to avoid the use of "magic numbers" which are hardcoded constants in the code. Instead, make sure to create a constant variable for it with a thought-out name. The function `BaseLoan._checkSignature()` uses the value `0x1626ba7e` to confirm a signature.
7. Mitigated Since the project applies some optimization techniques in several parts of the codebase, we would like to offer additional opportunities to make the code more efficient. The following is a non-exhaustive list of small optimizations:
    1. `AuctionLoanLiquidator.placeBid()` defines `max` but only uses it inside the `if` statement when throwing the error `AuctionOverError(max)`. Consider declaring it inside the `if` statement as done by the function `settleAuction()`.
    2. The `for` loops could be further improved by not initializing the index variable `i` to `zero` since this is already the default value. Furthermore, the access of an array's `length` field could be cached in a memory variable that then can be used within the condition.
    3. The variable `name` in `BaseLoan` can be immutable.

4. It is also convenient to compute some calculations once and reuse them instead of computing the same calculation multiple times. This is the case in `MultiSourceLoan._emitLoan()` where `offer.fee.mulDivUp(amount, offer.principalAmount)` is computed twice.
5. The variable `BaseLoan._liquidationAuctionDuration` is initialized twice to the same value of `3 days`. Consider removing the second initialization in the constructor.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

- `5d2...ade ./lib/AddressManager.sol`
- `eaf...020 ./lib/UserVault.sol`
- `9bd...017 ./lib/AuctionLoanLiquidator.sol`
- `ac9...23e ./lib/utils/Hash.sol`
- `9fb...c2a ./lib/utils/Interest.sol`
- `f38...a21 ./lib/callbacks/Leverage.sol`
- `0a0...d20 ./lib/loans/MultiSourceLoan.sol`
- `f77...57e ./lib/loans/BaseLoan.sol`
- `802...bb9 ./lib/loans/WithCallbacks.sol`

Tests

- `a75...25e ./test/AuctionLoanLiquidator.t.sol`
- `bd0...bf4 ./test/AddressManager.t.sol`
- `8ad...b69 ./test/UserVault.t.sol`
- `7bb...237 ./test/MultiSourceGas.t.sol`
- `188...1be ./test/TestNFTFlashAction.sol`
- `4a0...b74 ./test/validators/NftPackedListValidator.t.sol`
- `07f...20b ./test/validators/NftBitVectorValidator.t.sol`
- `03b...672 ./test/validators/RangeValidator.t.sol`
- `535...cbb ./test/utils/SampleMarketplace.sol`
- `59e...9bc ./test/utils/SampleToken.sol`
- `295...86c ./test/utils/SampleCollection.sol`
- `d1f...ae1 ./test/utils/USDCSampleToken.sol`
- `ccd...dc1 ./test/callbacks/Leverage.t.sol`
- `37e...ff9 ./test/loans/MultiSourceCommons.sol`
- `50e...000 ./test/loans/MultiSourceLoan.t.sol`
- `213...b86 ./test/loans/MultiSourceLoanTestExtra.t.sol`
- `6bd...c92 ./test/loans/TestLoanSetup.sol`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:
- Slither 0.10.0

Steps taken to run the tools:
1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

Slither found 506 results. These have either been classified as false positives or have been integrated into the findings of this report.

Test Suite Results

Tests were executed with `forge test`.

```
Running 3 tests for test/AddressManager.t.sol:AddressManagerTest
[PASS] testAddAndCheckentry() (gas: 532340)
[PASS] testConstructor() (gas: 460664)
[PASS] testRemoveFromWhitelist() (gas: 446571)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 91.60ms

Running 17 tests for test/callbacks/Leverage.t.sol:LeverageTest
[PASS] testAfterNFTTransferPunk() (gas: 359104)
[PASS] testBuy() (gas: 339388)
[PASS] testBuyFailInvalidCallback() (gas: 215149)
[PASS] testBuyFailOnlyWeth() (gas: 984759)
[PASS] testBuyFailWhitelist() (gas: 80798)
[PASS] testBuyPunk() (gas: 300614)
[PASS] testBuyReturn() (gas: 338895)
[PASS] testBuySeaport() (gas: 342524)
[PASS] testSell() (gas: 337873)
[PASS] testSellFailWhitelist() (gas: 274347)
[PASS] testSellInvalidCallback() (gas: 411976)
[PASS] testSellSeaport() (gas: 462413)
[PASS] testUpdateMultiSourceLoan() (gas: 1660552)
[PASS] testUpdateMultiSourceLoanFail() (gas: 1679046)
[PASS] testUpdateSeaport() (gas: 1660982)
[PASS] testUpdateSeaportFail() (gas: 1680804)
[PASS] testZeroAddress() (gas: 370591)
Test result: ok. 17 passed; 0 failed; 0 skipped; finished in 204.35ms

Running 2 tests for test/LiquidationDistributor.t.sol:LiquidationDistributorTest
[PASS] testLoanLiquidatedExcessProceeds() (gas: 4124091)
[PASS] testLoanLiquidatedLowerProceeds() (gas: 4092997)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 405.76ms

Running 17 tests for test/AuctionLoanLiquidator.t.sol:AuctionLoanLiquidatorTest
[PASS] testDoubleLiquidate() (gas: 134408)
[PASS] testGetAndSetDistributor() (gas: 22115)
[PASS] testLiquidateLoanNotAccepted() (gas: 107785)
[PASS] testLiquidateLoanNotOwner() (gas: 89974)
[PASS] testLiquidateLoanSuccess() (gas: 126902)
[PASS] testPlaceBidAuctionOver() (gas: 270769)
[PASS] testPlaceBidAuctionOverNoBids() (gas: 235167)
[PASS] testPlaceFirstBidNoAuction() (gas: 209894)
[PASS] testPlaceFirstBidSuccess() (gas: 236073)
[PASS] testPlaceSecondBidMinBidFail() (gas: 292478)
[PASS] testPlaceSecondBidSuccess() (gas: 303189)
[PASS] testSecondBidWithinMinMargin() (gas: 278676)
[PASS] testSettleNoBidsError() (gas: 133524)
[PASS] testSettleNotOverErrorBeforeExpiration() (gas: 244054)
[PASS] testSettleNotOverErrorNoMargin() (gas: 243650)
[PASS] testSettleSuccess() (gas: 285648)
[PASS] testZeroAddress() (gas: 244039)
Test result: ok. 17 passed; 0 failed; 0 skipped; finished in 37.29ms

Running 2 tests for test/validators/NftPackedListValidator.t.sol:NftPackedListValidatorTest
[PASS] testValidateOffer() (gas: 7004)
[PASS] testValidateOfferNoToken() (gas: 10600)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.37ms

Running 53 tests for test/loans/MultiSourceLoan.t.sol:MultiSourceLoanTest
[PASS] testEmitCollection() (gas: 223003)
[PASS] testEmitLoanInvalidBorrowerError() (gas: 131076)
[PASS] testEmitLoanInvalidBorrowerSignature() (gas: 140902)
[PASS] testEmitLoanInvalidLenderError() (gas: 155092)
```

```
[PASS] testEmitLoanInvalidLenderSignature() (gas: 113709)
[PASS] testEmitLoanOtherBorrowerCancelled() (gas: 167083)
[PASS] testEmitLoanOtherBorrowerExecuted() (gas: 336896)
[PASS] testEmitLoanPartialFillFail() (gas: 58275)
[PASS] testEmitLoanPartialFillSuccess() (gas: 217956)
[PASS] testEmitLoanSuccess() (gas: 223926)
[PASS] testEmitLoanSuccessWithBorrowerSignature() (gas: 258414)
[PASS] testEmitLoanSuccessWithLenderSignature() (gas: 235056)
[PASS] testEmitManyLoanSuccess() (gas: 350645)
[PASS] testEmitNotWhitelistedCallback() (gas: 159173)
[PASS] testEmitNotWhitelistedInvalidSelector() (gas: 191964)
[PASS] testEmitWithBorrowerOfferExpired() (gas: 52717)
[PASS] testEmitWithCallback() (gas: 252864)
[PASS] testEmitWithCallbackAndFeeTaxes() (gas: 367672)
[PASS] testEmitWithCanceledMinOffer() (gas: 78826)
[PASS] testEmitWithCanceledOffer() (gas: 78049)
[PASS] testEmitWithExecutedOffer() (gas: 246903)
[PASS] testEmitWithLenderOfferExpired() (gas: 53936)
[PASS] testEmitWithOverCapacity() (gas: 300108)
[PASS] testEmitZeroTokenId() (gas: 250485)
[PASS] testEmitZeroTokenIdInvalid() (gas: 132567)
[PASS] testExtendLockedSource() (gas: 5482510)
[PASS] testPartialRefinanceLockedSource() (gas: 5580465)
[PASS] testRefinanceCancelledAll() (gas: 3678173)
[PASS] testRefinanceCancelledOne() (gas: 3674880)
[PASS] testRefinanceFourSourcesMiddleOffer() (gas: 650012)
[PASS] testRefinanceFullBorrowerEqualPrincipal() (gas: 3840372)
[PASS] testRefinanceFullBorrowerLessPrincipal() (gas: 3839009)
[PASS] testRefinanceFullBorrowerMorePrincipal() (gas: 3899356)
[PASS] testRefinanceFullFailApr() (gas: 3660404)
[PASS] testRefinanceFullInvalidDuration() (gas: 3778031)
[PASS] testRefinanceFullInvalidPrincipal() (gas: 3776946)
[PASS] testRefinanceFullInvalidSignature() (gas: 3880443)
[PASS] testRefinanceFullStrict() (gas: 3792419)
[PASS] testRefinanceFullStrictInvalidSender() (gas: 3768854)
[PASS] testRefinanceInvalidHash() (gas: 239808)
[PASS] testRefinanceManySourcesOneLoan() (gas: 9121415)
[PASS] testRefinanceNewSourceTooSmall() (gas: 238425)
[PASS] testRefinancePartial() (gas: 288986)
[PASS] testRefinancePartialExpiredOffer() (gas: 240916)
[PASS] testRefinancePartialFailInvalidApr() (gas: 262161)
[PASS] testRefinancePartialFailInvalidLoan() (gas: 239281)
[PASS] testRefinancePartialFailNotStrictlyBetter() (gas: 243661)
[PASS] testRefinancePartialMultiple() (gas: 1762610)
[PASS] testRefinanceTooManySources() (gas: 4041825)
[PASS] testRefinanceWithFee() (gas: 5562843)
[PASS] testSetMinLockPeriod() (gas: 5223299)
[PASS] testUpdateImprovementMinimum() (gas: 5205425)
[PASS] testUpdateProtocolFee() (gas: 5287710)
Test result: ok. 53 passed; 0 failed; 0 skipped; finished in 2.02s

Running 2 tests for test/validators/RangeValidator.t.sol:RangeValidatorTest
[PASS] testValidateOffer() (gas: 5385)
[PASS] testValidateOfferOutOfRange() (gas: 11832)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.21ms

Running 23 tests for test/loans/MultiSourceLoanTestExtra.t.sol:MultiSourceLoanTestExtra
[PASS] testDelegate() (gas: 402547)
[PASS] testExecuteFlashAction() (gas: 253747)
[PASS] testExtendLoan() (gas: 245123)
[PASS] testExtendLoanMultipleSourcesError() (gas: 450738)
[PASS] testExtendLoanNotLenderError() (gas: 234052)
[PASS] testInvalidCallerFlashActionError() (gas: 228978)
[PASS] testLiquidationSingleSource() (gas: 219127)
[PASS] testMaliciousFlashAction() (gas: 372940)
[PASS] testOriginationAndRepayWithProtocolFee() (gas: 5500154)
[PASS] testRepayLoan() (gas: 307398)
[PASS] testRepayLoanWithSignature() (gas: 306236)
[PASS] testRepayLoanWithSignatureFail() (gas: 308910)
[PASS] testRepayLoanWithTaxAndFees() (gas: 372042)
[PASS] testRepayMany() (gas: 388481)
[PASS] testRepayWithCallback() (gas: 258836)
```

```
[PASS] testRepayWithNotWhitelistedCallback() (gas: 258234)
[PASS] testRepayWithWrongReturnCallback() (gas: 290308)
[PASS] testRevoke() (gas: 422590)
[PASS] testRevokeOutstandingError() (gas: 224870)
[PASS] testSendToLiquidation() (gas: 3919376)
[PASS] testSendToLiquidationNotExpired() (gas: 3910967)
[PASS] testSetDelegationRegistry() (gas: 20126)
[PASS] testSetFlashActionContract() (gas: 20704)
Test result: ok. 23 passed; 0 failed; 0 skipped; finished in 344.10ms

Running 2 tests for test/validators/NftBitVectorValidator.t.sol:NftBitVectorValidatorTest
[PASS] testValidateOffer() (gas: 5458)
[PASS] testValidateOfferNoToken() (gas: 9037)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.15ms

Running 24 tests for test/UserVault.t.sol:UserVaultTest
[PASS] testBurn() (gas: 79704)
[PASS] testBurnAndWithdraw() (gas: 313697)
[PASS] testBurnAndWithdrawNotApproved() (gas: 85351)
[PASS] testBurnNotApproved() (gas: 84678)
[PASS] testDepositERC20() (gas: 171346)
[PASS] testDepositERC20NotWhitelistedError() (gas: 84320)
[PASS] testDepositERC20WrongMethod() (gas: 80445)
[PASS] testDepositERC721() (gas: 148047)
[PASS] testDepositEth() (gas: 111998)
[PASS] testDepositManyNotWhitelistedError() (gas: 84891)
[PASS] testDepositMultipleNFTs() (gas: 216307)
[PASS] testDepositSingleNotWhitelistedError() (gas: 86360)
[PASS] testDepositVaultNotExists() (gas: 18861)
[PASS] testMint() (gas: 77148)
[PASS] testWithdrawAssetNotOwned() (gas: 209024)
[PASS] testWithdrawERC20() (gas: 206064)
[PASS] testWithdrawERC20NotReadyForWithdrawalNeverBurnt() (gas: 174761)
[PASS] testWithdrawERC20NotReadyForWithdrawalWrongRecipient() (gas: 188942)
[PASS] testWithdrawERC721() (gas: 150398)
[PASS] testWithdrawERC721NotReadyForWithdrawalNeverBurnt() (gas: 152389)
[PASS] testWithdrawERC721NotReadyForWithdrawalWrongRecipient() (gas: 147020)
[PASS] testWithdrawEth() (gas: 114705)
[PASS] testWithdrawEthNeverBurnt() (gas: 113603)
[PASS] testWithdrawEthWrongRecipient() (gas: 108072)
Test result: ok. 24 passed; 0 failed; 0 skipped; finished in 15.21ms

Running 6 tests for test/MultiSourceGas.t.sol:MultiSourceLoanTest
[PASS] testEmitMultiGas() (gas: 215795)
[PASS] testExtendLoanGas() (gas: 248332)
[PASS] testLiquidationGas() (gas: 24593294)
[PASS] testRefinanceFullWithManySourcesGas() (gas: 21673643)
[PASS] testRefinanceManyPartialSourcesGas() (gas: 3741381)
[PASS] testRepayGas() (gas: 20929752)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.51s

Ran 11 test suites: 151 tests passed, 0 failed, 0 skipped (151 total tests)
```

Code Coverage

The usual `forge coverage` command does not work, as the project requires the `--via_ir` flag to be activated, which is however always deactivated for the `coverage` command. There is an ongoing PR on this attempting to fix the issue, which was however not released in time for the audit.

Changelog

- 2023-10-20 - Initial report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

Notice of confidentiality

Links to other websites

Disclaimer

Quantstamp