PUBLIC

# Code Assessment

## of the Methlab Smart Contracts

March 11th, 2024

Produced for


by

# Contents

# 1 Executive Summary

Dear Methlab Team,

Thank you for trusting us to help Methlab with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Methlab according to Scope to support you in forming an opinion on their security risks.

Methlab implements an oracle-less lending protocol to arrange fixed-term loans between individual users. Moreover, a set of peripheral contracts is implemented to facilitate the interaction with the core protocol.

The most critical subjects covered in our audit are the safety of the funds in the lender vaults and the loans, the liveness of the protocol, and its functional correctness. A high-security issue was uncovered due to a dangling approval in the LoanExecutor smart contract where an attacker could steal a borrower's funds. Another high severity issue was uncovered during the second iteration of the codebase where a malicious borrower could grief the lender's assets. All the issues have been addressed.

The general subjects covered are access control, decentralization, documentation and specification, and testing. The specification is very extensive and the code is very well documented. The codebase is very well structured. The security of all aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 2 |
| • Code Corrected | 2 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 3 |
| • Code Corrected | 3 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Methlab repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 2 January 2024 | ee835c6f90b1cba8b0bb12ad7963cfff070dd9e0 | Initial Version |
| 2 | 15 January 2024 | 65a070a861cd25d5b85b67da2434990da646d5df | Fixes |
| 3 | 17 January 2024 | 09eec2eaaae7388cc929b262642b6723b0e13eed | Final fixes |
| 4 | 23 January 2024 | 3e356bbd9491dc30f95ea38bb31b79d3552d9e3a | Add MultiRoute |
| 5 | 11 March 2024 | 8e060852978aafaa62dbbf49c5889dcdb5f91 | Renaming |

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

The following files are in scope. SHA256 hashes are provided for convenience.

```
4fa413eb77650e3543c1f85a250c0bf3007539faef5ebdcbf6066b7bf18ffbce  packages/contracts/src/Factory.sol
790042fb431903b31932b6b4aeb5dc4b52732820e149f1aed0339c975a96909c  packages/contracts/src/interfaces/IERC20Decimals.sol
e5bf1c779f1a2d89375acd1dd017edaf561057fa98e652ac80ef02138f1a5a89  packages/contracts/src/interfaces/IFactory.sol
284095b36e514b8e01d14164b6a1616f842330915f08dc972eaf3f418f622e0b  packages/contracts/src/interfaces/ILendersVault.sol
662a8c636fa59069bbc1d82143dd4168c07ec74ba1ab564b65bac1c9164c8a0b  packages/contracts/src/interfaces/ILoan.sol
e0de9777e77176f57b0e20a7fcb3f4ddc8eb894895aeee711e55c7b2ae4980b9  packages/contracts/src/interfaces/ILoanTaker.sol
1092b2448cb2f5f4aa1f93f9003eb0956c108855d65ad38e2297b372a2451009  packages/contracts/src/interfaces/IRegistry.sol
f0b9e400e51266a116dcf360dc50630ff634efbc7a588cedbca332d7d40b9bc0  packages/contracts/src/interfaces/IRouter.sol
4a5f7fdad7e997f5381b418265f8b2fb7e66606e35e360d383ac48c42b009916  packages/contracts/src/interfaces/ISwapper.sol
ce8f3e2d192d5c07cc1de3d9d03bc84b9166aa29ac18ab2a416c67b5ea705c51  packages/contracts/src/LendersVault.sol
1b7ff55177011227d5e37bd6b2e9e84200fc884e521d1cb0f56fb861729b0bb2  packages/contracts/src/libraries/DataTypes.sol
4f4701429a522a29d79692225138650bf2f96da62729ad90d1fcea658ed65a6d  packages/contracts/src/libraries/Errors.sol
226ff1d90a5ee64b48925d86cd032d1f49bb48cf1466baefe072e26bb0a7df31  packages/contracts/src/libraries/Helper.sol
b48e3ff73e7d6e29adaf60341def1ab730f4d8abb4973876b1f87afb560ecd04  packages/contracts/src/libraries/UniswapV2Library.sol
99ccc00879bda0d16f1233fa4d195958f5070cfd1073ae8aa7d0ba2bd80a091c  packages/contracts/src/Loan.sol
f471bb401e4c8de794ca610e992511e17709d71276c2c06e2182465d4e5cb247  packages/contracts/src/periphery/LoanExecutor.sol
c34e8016e492236c426b1fe0ac1e4e9713fb7d78296c238cf3a7ee85f6c04a08  packages/contracts/src/periphery/UniswapV2Swapper.sol
b53079fd25b99054dfe6aa3846c6c786d33acfe79295756084a9c3e8d0174c01  packages/contracts/src/periphery/UniswapV3Swapper.sol
de176bd5ebfad592ed960e00f195649b85e55208037dbb642bfddbb8b061e75d  packages/contracts/src/periphery/UniswapV3SwapperMultiRoute.sol
c0d4bb9c7bad0adb341097d3bbac618b193cf9556d05ba341a1bcec97750871c  packages/contracts/src/Registry.sol
```

### 2.1.1 Excluded from scope

Any file not included in Scope is out-of-scope. In particular, third-party libraries, including patched versions, and deployment scripts are not in scope. The evaluation of the economic model of the implemented protocol is beyond the scope of this review. The protocol doesn't implement any restrictions on the configurations of the various modules such as the Lender's vault. Therefore, lenders are responsible for configuring their vaults and borrowers are responsible for correctly choosing the lender vaults from which they'll take loans. The owner of the protocol can set the protocol fees. Attacks concerning a malicious owner were considered out of scope. The smart contracts are aimed to be deployed on the Mantle L2 rollup. We assume that the semantics of the EVM opcodes are the same as on Ethereum. Moreover, the Mantle sequencer is considered to be fully trusted and to function as expected.

## 2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Methlab offers an oracle-less protocol to arrange fixed-term loans between individual users.

Lenders publish *intents* indicating their willingness to lend some fungible asset, the *principal*, with another fungible asset as *collateral*, with a certain *strike price*, *interest rate*, and *duration*. The strike price determines the amount of principal lent per unit of collateral. Each loan has has a duration and an expiration day. The lender cannot liquidate the borrower while the loan is ongoing: instead, the borrower can choose to partially or fully *repay* the loan before it expires to receive a proportional amount of their collateral. If they choose not to, the lender can *seize* the remaining collateral after the expiration date. As a result, the loan replicates an American option, allowing the borrower to buy back the collateral at the strike price before expiry.

### 2.2.1 Factory

This singleton contract deploys LendersVault and Loan contracts using immutable proxies. It is responsible for initializing the new contracts and adding them to its Registry. Anyone can deploy a vault, but only vaults can deploy loans.

### 2.2.2 Registry

This singleton contract maintains a list of all valid LendersVault and Loan instances. It only allows its Factory to register contracts.

### 2.2.3 LendersVault

This contract can be deployed as an immutable proxy. It is owned by a lender and allows borrowers to take loans with that lender.

To determine what loans are possible, the owner must create *intents*, described later, and *intent collections*.

**Intent Collections:**

They specify a collateral and borrowable token, the minimum and maximum amounts for a single loan, an expiration timestamp for the collection, and an `isEnabled` flag which determines whether new loans can be issued using this collection. Once expired, no further loans can be issued through a collection, but existing loans are not affected. It is possible to change the expiration date (`extendIntentCollection()`) of a collection or to toggle `isEnabled` (`setIntentCollectionStatus()`) . The new expiration date is required to be in the future. The rest of the fields are immutable.

**Intents:**

Intents specify a strike price, interest rate, and duration. They belong to exactly one intent collection. An intent is created together with its collection by calling `createIntentCollection()` and is immutable. If an intent belongs to a collection that is neither expired nor disabled, any borrower can call `createLoan()` referencing it.

The LendersVault exposes the following interface:

- `createLoan()`: It requires a collection id, intent id, and a borrow amount. A user has the option to directly convert the principal tokens borrowed to collateral tokens. For that case, callback data for `sourceCollateral()` can also be provided. `createLoan()` will compute

the principal amount as the collateral amount multiplied by the strike price, ensure that it is within the collection bounds, and transfer the principal amount in borrow tokens to the caller. It then deploys a `Loan` instance and sets the borrower to the caller, the borrowed amount to the principal amount augmented with the interest, and the expiration to the current timestamp plus the collection duration. Then, if callback data was provided, it calls back the caller's `sourceCollateral()` method passing the callback data. Finally, it tries to transfer the collateral from the caller, reverting if this fails.

- `withdraw()`: In order to provide principal, the owner of a vault must manually transfer tokens into the contract. The `withdraw()` function allows the owner to transfer any tokens held by the contract back to themselves.
- `set[Un]pauseGuardian()`: The owner can designate a new [un]pause guardian address.
- `[un]pause()`: The owner or the designated guardian can [un]pause the contract, disallowing the creation of new loans.

### 2.2.4 Loan

This contract can be deployed as an immutable proxy. It sits between a lender and a borrower and holds the collateral.

The Loan contract exposes the following interface:

- `repay()` As long as the loan hasn't expired, anyone can call the `repay()` function to reduce the loan amount. It requires an amount in the principal token and some optional callback data. If the amount is nonzero and less than or equal to the remaining debt amount, the accounting is updated, then the released amount of the collateral is transferred to the borrower. Then, if callback data was provided, the `sourcePrincipal()` is called on the caller, passing the data. The callback allows to convert of part of the borrower's collateral into principal. Finally, the amount to be repaid is transferred from the caller.
- `transfer()`: The borrower can `transfer()` the loan to a different address. The collateral is always refunded to the current borrower's address.
- `seizeCollateral()`: After the loan is expired, the `seizeCollateral()` function can be called by anyone. The function refunds the lender vault address with the remaining collateral based on internal accounting.

### 2.2.5 LoanExecutor

This periphery contract is a singleton. It allows users to take and manage trading positions using loans. It does not hold tokens but receives ERC-20 allowance from users.

The LoanExecutor exposes the following interface:

- `executeLoans()`: The function takes an array of loan parameter elements. Each element specifies a lender's vault, an amount of collateral to swap, a lender intent, an amount of collateral to supply, and optionally the address of a swapper contract and some callback data. For each element, the contract will consult the lender's intent to discover the principal and collateral tokens. Then it will transfer the difference between the amount of collateral to supply and the amount of collateral to swap from the user. Then it will give allowance to the vault contract for the collateral and call `createLoan()`. If a swap is needed, a `sourceCollateral()` callback will occur, allowing the LoanExecutor to swap some of the principal received to collateral using the specified swapper. Finally, the function transfers the loan and any principal or collateral amount left to the caller and proceeds with the next element.

- `repayLoan()` The function takes the address of a loan, the address of a swapper contract, a repayment amount, and some callback data for the swapper. It queries the loan contract to

discover the tokens involved. Then, it gives allowance in the borrow token to the loan and calls `repay()`. This results in a `sourcePrincipal()` callback where the freed collateral is transferred from the borrower and swapped into the amount of borrow token required for repayment. Finally, the remaining collateral is given back to the sender.

### 2.2.6 UniswapV2Swapper

This contract is an adapter for LoanExecutor to trade on Uniswap version 2. It contains two functions, `swapIn()`, which swaps an amount of input tokens already transferred to the contract and routes the resulting funds back to the caller, and `swapOut()`, which swaps as few input tokens as possible to obtain a specified amount of output tokens and transfers the remainder back to the caller. Both functions use price information passed by the frontend to prevent sandwich attacks.

### 2.2.7 UniswapV3Swapper

This contract is an adapter for LoanExecutor to trade on Uniswap version 3. Its semantics are the same as UniswapV2Swapper which we refer to.

### 2.2.8 Trust model

The singleton contracts in the protocol are immutable and fully permissionless.

Once a LendersVault is deployed, its owner is in control of all assets deposited therein. It should ensure that any seizable collateral is seized promptly to minimize opportunity costs. If an intent is enabled, anyone can use it to borrow. The owner can designate other addresses that it lightly trusts as pause and unpause guardians. They are only able to call the `pause()` and `unpause()` functions respectively. Ownership cannot be transferred.

Once a Loan is deployed, anyone can call `repay()`, but the borrower will receive the collateral. Anyone can call `seizeCollateral()`, but the collateral will be transferred to the lender's vault. The borrower is incentivized to ensure repayment happens if it is profitable to them. They have no special privileges on the contract besides transferring ownership.

### 2.2.9 Changes in Version 2

- The specification of `extendIntentCollection()` has been changed. It is allowed to shrink the duration of an intent collection as long as the expiry is set to the future.
- The meaning of interest rate has been changed. Originally, the interest rate was the percentage of the borrowed amount that should be returned together with the borrowed amount. For a loan of 1000$ with 10% interest an extra 100$ should be returned to the system by the borrower. In V2, is the percentage of the loan the borrower doesn't actually receive. This means, ignoring taker fees, for a loan of 1000$ with 10% interest, the borrower will only receive 900$ but they should return 1000$ in the end.
- A fee system was introduced. Both makers and takers should pay a fee (`makerFee`, `takerFee`). The fees are determined arbitrarily by the owner of the Registry contract. More specifically:
    - `takerFee`: It's calculated as a percentage of the premium the borrower needs to pay which is added on top of the premium. The fee is initially deducted from the lender's vault balance but it is expected to be returned if it makes sense for the borrower to repay the loan.
    - `makerFee`: It's calculated as a percentage of the premium. It's withdrawn from the lender's vault.
- The lenders are expected to take into account the fee specification to correctly specify reasonable strike prices.

### 2.2.10 Changes in Version 4

In Version 4, UniswapV3SwapperMultiRoute has been added to support multi-hop swaps.

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 2 |

- Incorrect Seizable Amount Calculation Code Corrected
- Borrower Profits Can Be Stolen Code Corrected

| | |
|---|---|
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 3 |

- Expiry Extension Code Corrected Specification Changed
- Extending and Enabling Collection With Id 0 Code Corrected
- Preview Repay Code Corrected

| | |
|---|---|
| Informational Findings | 1 |

- Dead Code Code Corrected

## 6.1 Incorrect Seizable Amount Calculation

Correctness High Version 2 Code Corrected

*CS-METHLAB-008*

In `Loan.seizeCollateral()`, the seized collateral amount is computed like so:

```
(, uint256 reclaimableAmt) = previewRepay(loanData.repaidAmt);
uint256 seizeableAmt = loanData.collAmt - reclaimableAmt;
```

In version 2, the `Loan.previewRepay()` functions is as follows:

```
function previewRepay(uint256 repayAmt) public view returns (uint256, uint256) {
  uint256 remainingAmt = loanData.borrowAmt - loanData.repaidAmt;
  if (repayAmt > remainingAmt) {
    repayAmt = remainingAmt;
  }
  return (repayAmt, (loanData.collAmt * repayAmt) / loanData.borrowAmt);
}
```

Because of the `repayAmt > remainingAmt` check, `seizeCollateral` ends up overestimating the available collateral to seize if more than half of the loan has been repaid, thereby causing the seizure to fail. This essentially griefs the lender via denial of service.

For instance, assume the following parameters:

```
loanData.borrowAmt = 1000
loanData.repaidAmt = 600
```

Then, the function will compute:

```
repayAmt = loanData.repayAmt == 600
remainingAmt = loanData.borrowAmt - loanData.repaidAmt == 400
(repayAmt > remainingAmt) == true
repayAmt = remainingAmt == 400
reclaimableAmt = (loanData.collAmt * repayAmt) / loanData.borrowAmt = 40% of loanData.collAmt
seizeableAmt = loanData.collAmt - reclaimableAmt = 60% of loanData.collAmt
```

However, the correct value for `seizableAmt` would be 40% of the collateral.

---

**Code corrected:**

The collateral is now seized simply transferring the balance in collateral of the loan contract back to the lender's vault.

## 6.2 Borrower Profits Can Be Stolen

Security | High | Version 1 | Code Corrected

*CS-METHLAB-005*

Methlab implements the `LoanExecutor` periphery contract to facilitate loan execution and repayment. `LoanExecutor` implements the `sourcePrincipal` fallback function which can be used to convert some of the collateral amount the borrower holds into principal so it can be repaid for the loan. For that, a borrower must give a token allowance to an instance of `LoanExecutor`. If the borrower holds an option that is in profit, then any attacker can repay their loan through the executor and pocket their profit.

Let us consider the following scenario:

1. A lender offers loans with:
    - Collateral: `mETH`
    - Principal: `mUSD`
    - Strike price: 1500 `mUSD/mETH`
    - Market price: 2000 `mUSD/mETH`
    - Interest rate: 1% (for 30 days)
    - Duration: 30 days
2. A borrower/victim `V` creates a loan by offering 1 `ETH` as collateral, receiving 1500 `mUSD`. The borrower must repay 1515 `mUSD` to receive back the collateral.
3. `V` being short on `mUSD` converts it to `mETH` at market price receiving 0.75 `mETH`.
4. After some time the `mETH` price goes up to 4000 `mUSD/mETH`.
5. `V` decides to repay their loan using the `LoanExecutor` (`LE`) and they give an approval to it.
6. An attacker `A` sees the approval and calls `LE.repayLoan()` before `V` does so. The attacker tries to repay the full loan i.e., `repayAmt` is 1515 `mUSD`.
7. `repayLoan` makes a call to `Loan.repay()` which eventually calls `LE.sourcePrincipal()`.
8. `LE` withdraws 1 ETH from the borrower. Since the price of `mETH` is now 4000, only a part of the 1 will be used to obtain the needed 1515 `mUSD`. The rest of the `mETH` is returned to the `LE`.
9. At the end of the call the `LE` returns the remaining `mETH` to the `msg.sender`.

```
/// @dev transfer any remaining collToken back to borrower
transferRemaining(collToken, msg.sender);
```

However, the assets are not owned by the `msg.sender` who's the attacker and not the borrower.

Moreover, when repaying, the executor could sell the collateral at the strike price or better. The attacker can force the executor to trade with them at the strike price by passing a custom `swapper` contract, or by manipulating Uniswap with sandwich calls. If the collateral has increased in value, this is profitable at the expense of the borrower.

---

**Code corrected:**

A loan can only be repaid by the borrower. Hence, an attacker cannot take advantage of dangling approvals.

## 6.3 Expiry Extension

Design | Low | Version 1 | Code Corrected | Specification Changed

*CS-METHLAB-006*

Owners of a vault can call `extendIntentCollection()` to extend the expiration date of a collection. However, the expiration date can be set in timestamp earlier than what it currently is, essentially shrinking the lifetime of the collection.

---

**Code corrected:**

The new expiration date must be in the future.

**Specification changed:**

The specification delivered to us by Methlab now allows it to be earlier than the previously set expiration date.

## 6.4 Extending and Enabling Collection With Id 0

Design | Low | Version 1 | Code Corrected

*CS-METHLAB-007*

A new collection is assigned to an id. This id is strictly greater than 0 meaning there's no collection with id 0. However, the owner of a lender's vault can successfully call `extendIntentCollection()` and `setCollectionStatus()` for the collection with id 0.

---

**Code corrected:**

The aforementioned methods will revert for collections with id 0.

## 6.5 Preview Repay

Design | Low | Version 1 | Code Corrected

*CS-METHLAB-009*

`Loan.previewRepay()` is used to preview how much collateral will be repaid for a given principal amount (`repayAmt`). However, there's no check that the `repayAmt` doesn't exceed the total borrowed amount. Therefore a caller of this function (EOA or smart contract) could accidentally assume that they can withdraw more collateral than they are supposed to.

---

**Code corrected:**

The `previewRepay()` function returns two values: 1) the amount of the collateral to be released for a certain amount principal 2) the capped amount of the principal to be repaid.

## 6.6 Dead Code

**Informational** **Version 1** **Code Corrected**

*CS-METHLAB-004*

The `swapIn()` function in the `UniswapV2Swapper` and `UniswapV3Swapper` contracts is never used within the repo.

---

**Code corrected:**

Dead code was removed

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 Gas Optimizations

**Informational** **Version 1**

*CS-METHLAB-001*

We report a non-exhaustive list of potential gas optimizations.

1. In Factory, the existence of a vault or loan is registered with a boolean. Since solidity performs additional masking operations when storing booleans, it would be more efficient to use a uint256.
2. In periphery contracts, `safeIncrease-` and `decreaseAllowance()` are used to set the approval to a specific value and later fully revoke it. However, the alternatives `forceApprove()` and `safeApprove()` could be used to increase clarity.
3. In LoanExecutor, the function `createLoan()` could be public, making creating a single loan slightly cheaper.
4. `block.timestamp+1` is used as the deadline for UniswapV2 and UniswapV3 routers, which is unnecessary and can be changed to `block.timestamp`.

## 7.2 No Sanity Checks

**Informational** **Version 1**

*CS-METHLAB-002*

`LendersVault.createIntentCollection` doesn't enforce any sanity checks on some of the input parameters. For example:

- `minSingleLoanAmt` is not checked to be less than `maxSingleLoanAmt`
- `borrowToken` is not checked to be different from `collToken`

These sanity checks do not affect the security of the protocol just the user experience for the lenders since badly parameterized intents will not result in loans.

`UniswapV3SwapperMultiRouter` does not ensure that `path[0] == tokenIn` and `path[n] == tokenOut`. In case the output token encoded in the path is not as expected, a call may still succeed if there is sufficient `tokenOut` in `LoanExecutor` (e.g. via a donation). And the erroneous output tokens will be locked. These tokens can be unlocked via another loan through the `LoanExecutor` if this loan handles these tokens.

## 7.3 Redundant Events

**Informational** **Version 1**

*CS-METHLAB-003*

Some admin functions in `LendersVault` emit redundant events even if the new value is the same as the old one. These functions are:

- `extendIntentCollection()`
- `setIntentCollectionStatus()`
- `set[Un]pauseGuardian()`

# 8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

## 8.1 Interest Is Not Accrued Over Time

Note Version 1

Typically, when the term interest is used, it is understood that if a lender repays early, they will be charged less interest. In Methlab, this is not the case at the scale of a single loan, since the borrower is always charged the full interest amount regardless of when they repay. Lenders should be aware that borrowers are not directly incentivized to repay early.

## 8.2 Potential Read-Only Reentrancy

Note Version 1

A user can repay a loan by calling `Loan.repay()`. During the call, they can optionally make a call back to the msg.sender, should non-empty data have been passed. Note that at the point of the call, the state of the Loan has not been fully updated as collateral has been transferred to the borrower but no principal has been transferred from the msg.sender. An external contract assuming that the `Loan` contract is always in a consistent state could accidentally read the balances and malfunction.

In Version 2, a public function `reentrancyGuardEntered()` function has been added to allow other contracts to query the state of the Loan's reentrancy guard.

## 8.3 Properly Setting the Strike Price

Note Version 1

An intent specifies a specific strike price for a loan. Of course, an intent configuration can become non-profitable for the lender if the market price of either the collateral or the principal changes. In that case, the lender can disable the intent. Lenders should set the strike price in such a way that they have sufficient time to react to rapid price changes in the market.

## 8.4 Protocol Fees Can Be Set Arbitrarily

Note Version 2

The administrator of the protocol controls the quantity of taker and maker fees charged by the protocol. A call to the setter methods can affect all existing loan intents immediately, and both fees can be set to 100% of the loan premium. Lenders and borrowers should make sure they trust the administrator address.

## 8.5 Rounding Errors on Repayment

Note Version 1

When repaying a loan, `previewRepay` calculates the amount to be reclaimed to the user based on the ratio of the repaid amount over the total borrowed amount. This division can introduce some rounding errors. These errors are in favor of the lender who can, in theory, seize all the remaining balance of the collateral after the loan has expired. A theoretical attack could be the following: A malicious lender gradually repays very tiny amounts of the loan. As the collateral to be released is rounded down to zero and the repaid amount is sent back to the lender, the attacker only loses on gas fees. If the loan is repaid, the borrower cannot claim the collateral. When the loan expires the lender can seize the collateral.

## 8.6 Unsupported ERC-20 Tokens

**Note** **Version 1**

The system doesn't apply any restrictions on what tokens are used as collateral and principal tokens. However, it is important to emphasize that tokens with rebases should not be used as collateral. Moreover, tokens with negative rebases or fees on transfer should not be used as either collateral or principal. Lenders should be mindful of the tokens they choose to allow in their intents.

A token address with no contract code should also not be used as collateral since *safeTransferFrom()* calls will always succeed. As a result, the borrow token can be drained.