Lending

Smart Contract Audit Report
Prepared for MoneySwitch

Date Issued:	Sep 22, 2022
Project ID:	AUDIT2022043
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022043
Version	v1.0
Client	MoneySwitch
Project	Lending
Auditor(s)	Natsasit Jirathammanuwat Ronnachai Chaipha Wachirawit Kanpanluk
Author(s)	Natsasit Jirathammanuwat Ronnachai Chaipha Wachirawit Kanpanluk
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 22, 2022	Full report	Natsasit Jirathammanuwat Ronnachai Chaipha Wachirawit Kanpanluk

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary **1**
- 1.1. Audit Result 1
- 1.2. Disclaimer 1

2. Project Overview **2**
- 2.1. Project Introduction 2
- 2.2. Scope 3

3. Methodology **6**
- 3.1. Test Categories 6
- 3.2. Audit Items 7
- 3.3. Risk Rating 9

4. Summary of Findings **10**

5. Detailed Findings Information **13**
- 5.1. Lack of Payment Source Authorization Check 13
- 5.2. Loan Interest Miscalculation 17
- 5.3. Improper Withdrawal Amount 21
- 5.4. Signature Reutilization 26
- 5.5. Centralized Control of State Variable 30
- 5.6. Insufficient Check for Approved Operation 32
- 5.7. Improper Inactive Pools Handling 35
- 5.8. Reentrancy Attack 40
- 5.9. Borrowing Credit Interest Not Included 43
- 5.10. Loan Repayment Date Not Enforced 48
- 5.11. Insufficient Liquidation Flow Check 51
- 5.12. Business Design Flaw 55
- 5.13. Improper Feeder Pool Count Increment 59
- 5.14. Missing Input Validation 63
- 5.15. Repay Interest Miscalculation 66
- 5.16. Integer Underflow 69
- 5.17. Insufficient Logging for Privileged Functions 73
- 5.18. Unsafe Token Transfer 76
- 5.19. Inexplicit Solidity Compiler Version 79

6. Appendix **81**
- 6.1. About Inspex 81

1. Executive Summary

As requested by MoneySwitch, Inspex team conducted an audit to verify the security posture of the Lending smart contracts between Aug 15, 2022 and Aug 24, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Lending smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 3 high, 11 medium, 1 low, 2 very low, and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that Lending smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

MoneySwitch is a liquidity-as-a-service (LaaS) DeFi lending platform. MoneySwitch provides cross-border payment companies access to real-time liquidity to process cross-border payments and allows lenders to earn competitive yields by lending to some of the world's largest cross-border payment providers.

MoneySwitch is an uncollateralized lending protocol on the Ethereum blockchain which establishes a liquidity pool for the purpose of borrowing and lending assets.

Scope Information:

Project Name	Lending
Website	https://moneyswitch.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum Mainnet
Programming Language	Solidity
Category	Lending

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 15, 2022 - Aug 24, 2022
Reassessment Date	Sep 20, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 437ec0e5d9c17e9173fc87629b419f97e21a1ac3)

Contract	Location (URL)
DataVault	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/DataVault.sol
Vaultable	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/Vaultable.sol
Governance	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/access/Governance.sol
Multiownable	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/access/Multiownable.sol
BorrowerRegistry	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/borrower/BorrowerRegistry.sol
CreditProtectionPool	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/credit_protection/CreditProtectionPool.sol
LoanRegistry	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/loan/LoanRegistry.sol
LoanWallet	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/loan/LoanWallet.sol
InterestRateCalculator	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/master_lender/InterestRateCalculator.sol
MasterLender	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/master_lender/MasterLender.sol
MasterLiquidator	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/master_liquidator/MasterLiquidator.sol
FeederPool	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/pool/FeederPool.sol
MasterPool	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/pool/MasterPool.sol
PoolRegistry	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/pool/PoolRegistry.sol

RevenueDistribution	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/revenue_distribution/RevenueDistribution.sol
MasterRewards	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/rewards/MasterRewards.sol
RewardLocker	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/rewards/RewardLocker.sol
DeveloperTreasury	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/treasuries/DeveloperTreasury.sol
DistributionTreasury	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/treasuries/DistributionTreasury.sol
Treasury	https://github.com/moneyswitch/ms-contracts/blob/437ec0e5d9/contracts/treasuries/Treasury.sol

Reassessment: (Commit: 48f7a872353c957809239676615e0920d3eb3b95)

Contract	Location (URL)
DataVault	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/DataVault.sol
Vaultable	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/Vaultable.sol
Governance	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/access/Governance.sol
Multiownable	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/access/Multiownable.sol
BorrowerRegistry	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/borrower/BorrowerRegistry.sol
CreditProtectionPool	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/credit_protection/CreditProtectionPool.sol
LoanRegistry	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/loan/LoanRegistry.sol
LoanWallet	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/loan/LoanWallet.sol
InterestRateCalculator	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/master_lender/InterestRateCalculator.sol
MasterLender	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/mas

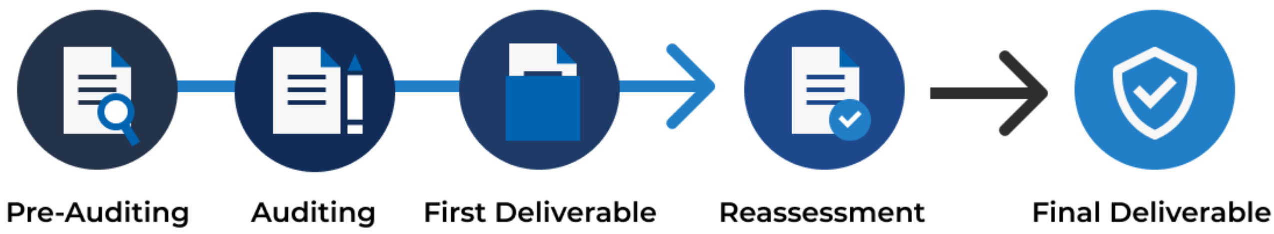
	ter_lender/MasterLender.sol
MasterLiquidator	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/master_liquidator/MasterLiquidator.sol
FeederPool	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/pool/FeederPool.sol
MasterPool	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/pool/MasterPool.sol
PoolRegistry	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/pool/PoolRegistry.sol
RevenueDistribution	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/revenue_distribution/RevenueDistribution.sol
MasterRewards	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/rewards/MasterRewards.sol
RewardLocker	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/rewards/RewardLocker.sol
DeveloperTreasury	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/treasuries/DeveloperTreasury.sol
DistributionTreasury	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/treasuries/DistributionTreasury.sol
Treasury	https://github.com/moneyswitch/ms-contracts/blob/48f7a87235/contracts/treasuries/Treasury.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

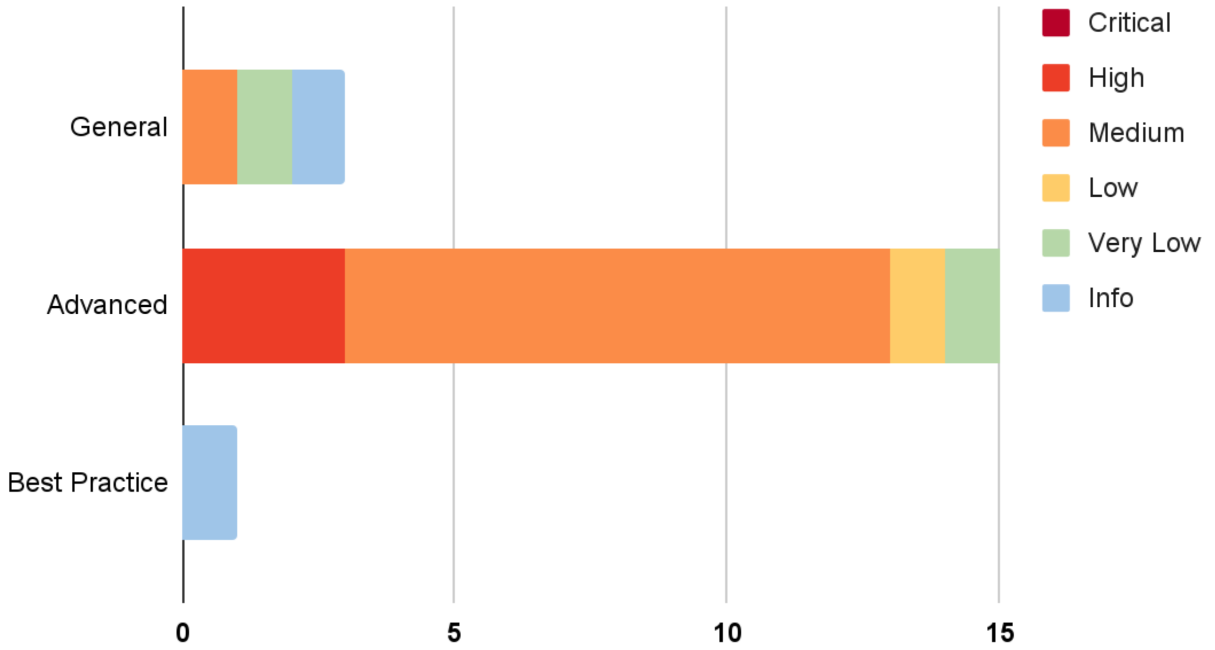
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

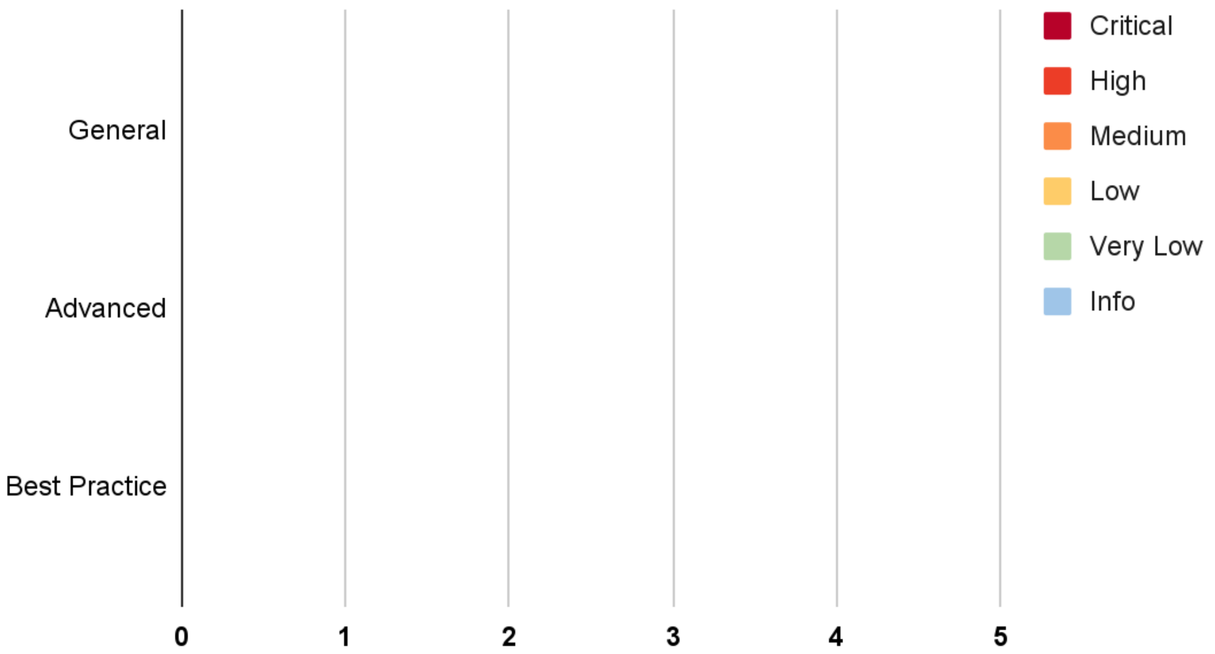
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Lack of Payment Source Authorization Check	Advanced	High	Resolved
IDX-002	Loan Interest Miscalculation	Advanced	High	Resolved
IDX-003	Improper Withdrawal Amount	Advanced	High	Resolved
IDX-004	Signature Reutilization	Advanced	Medium	Resolved
IDX-005	Centralized Control of State Variable	General	Medium	Resolved
IDX-006	Insufficient Check for Approved Operation	Advanced	Medium	Resolved
IDX-007	Improper Inactive Pools Handling	Advanced	Medium	Resolved
IDX-008	Reentrancy Attack	Advanced	Medium	Resolved
IDX-009	Borrowing Credit Interest Not Included	Advanced	Medium	Resolved
IDX-010	Loan Repayment Date Not Enforced	Advanced	Medium	Resolved *
IDX-011	Insufficient Liquidation Flow Check	Advanced	Medium	Resolved *
IDX-012	Business Design Flaw	Advanced	Medium	Resolved *
IDX-013	Improper Feeder Pool Count Increment	Advanced	Medium	Resolved
IDX-014	Missing Input Validation	Advanced	Medium	Resolved
IDX-015	Repay Interest Miscalculation	Advanced	Low	Resolved
IDX-016	Integer Underflow	Advanced	Very Low	Resolved
IDX-017	Insufficient Logging for Privileged Functions	General	Very Low	Resolved

IDX-018	Unsafe Token Transfer	General	Info	Resolved
IDX-019	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

* The mitigations or clarifications by MoneySwitch can be found in Chapter 5.

5. Detailed Findings Information

5.1. Lack of Payment Source Authorization Check

ID	IDX-001
Target	MasterLender
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High If the user has already approved tokens to the contract for repaying the loan, the attacker can drain the user's tokens from the wallet to repay any loan on the platform. **Likelihood:** Medium Anyone can call the `repay()` function but the victim has to approve tokens to the contract and the fund will be repaid to the existing loan, which can only be created by the registered borrower by the platform owner.
Status	Resolved The MoneySwitch team has resolved this issue by changing the payment source to the `msg.sender` in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.1.1. Description

The `repay()` function in the `MasterLender` contract gets the `paymentSource_` as an input, as shown in line 124. The `paymentSource_` parameter is then passed to the `_repayPartial()` and `_repayFull()` functions in lines 151 and 154.

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
127 ) external {
128     // Get Loan by ID
129     Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);
130 
131     // Check if loan is active
132     require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");
133 
134     // Get current timestamp
135     uint256 ts_ = block.timestamp;
```

```

    // Get total outstanding interestRounded
    uint256 interestRounded_ = _calculateInterestRounded(
        loan_,
        loan_.currentPrincipal,
        ts_
    ) + loan_.interestLocker;

    // Get interest on repayment principal
    uint256 interest_ = _calculateInterest(loan_, amount_, ts_);

    // If the amt is smaller than than current principal + interestRounded then it is
    // partial payment, otherwise it is a full payment
    if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
        // Make partial principal repayment
        _repayPartial(paymentSource_, loan_, amount_, interest_);
    } else {
        // Make full payment
        _repayFull(paymentSource_, loan_, interestRounded_, ts_);
    }
}
```

At lines 259 and 311, in the `_repayFull()` and the `_repayPartial()` functions, they both call the `_makePayment()` function with the `paymentSource_` parameter.

MasterLender.sol

```
243 function _repayFull(
244     address paymentSource_,
245     Loan memory loan_,
246     uint256 interestRounded_,
247     uint256 ts_
248 ) private {
249     uint256 totalInterest_ = _calculateInterest(
250         loan_,
251         loan_.currentPrincipal,
252         ts_
253     ) + loan_.interestLocker;
254
255     // Calculate Slippage
256     uint256 slippage_ = interestRounded_ - totalInterest_;
257     // Make payment
258     _makePayment(
259         paymentSource_,
260         loan_.currentPrincipal,
261         totalInterest_,
262         slippage_
```

```
    );

    // Clear the interest locker
    dataVault.loanRegistry().resetInterestLocker(loan_.wallet);

    // Set loans state to paid
    dataVault.loanRegistry().setStatus(loan_.wallet, LoanStatus.Paid);

    // Calibrate contracts
    _calibrateRepayment(loan_, loan_.currentPrincipal);

    // Emit Payment Event
    emit Repaid(
        loan_.wallet,
        paymentSource_,
        (loan_.currentPrincipal + totalInterest_ + slippage_)
    );
}
```

MasterLender.sol

```
function _repayPartial(
    address paymentSource_,
    Loan memory loan_,
    uint256 amount_,
    uint256 interest_
) private {
    // Payment Variables
    uint256 principalPayment_ = amount_;
    uint256 interestPayment_ = 0;

    // If the amount_ is larger than principal then it covers the full currentPrincipal
    // and it covers partial payment of interest
    if (amount_ > loan_.currentPrincipal) {
        principalPayment_ = loan_.currentPrincipal;
        interestPayment_ = amount_ - loan_.currentPrincipal;
    }
    // Store outstanding interest, extract any paidup interest
    dataVault.loanRegistry().increaseInterestLocker(
        loan_.wallet,
        (interest_ - interestPayment_)
    );

    // Make Payment
    _makePayment(paymentSource_, principalPayment_, interestPayment_, 0);

    // Calibrate other contract according to repayment amount
```

```
    _calibrateRepayment(loan_, principalPayment_);

    // Emit Event
    emit Repaid(loan_.wallet, paymentSource_, amount_);
}
```

The `_makePayment()` function will transfer the token from the `paymentSource_` address to the `RevenueDistribution` contract to distribute the funds. Since the token is transferred by using the `transferFrom()` function in line 335, the `paymentSource_` address has to approve tokens to the `MasterLender` contract.

MasterLender.sol

```
327 function _makePayment(
328     address paymentSource_,
329     uint256 principal_,
330     uint256 interest_,
331     uint256 slippage_
332 ) private {
333     // Move principal and interest to revenue distribution pool
334     _pool.liquidityAsset().transferFrom(
335         paymentSource_,
336         address(dataVault.revenueDistribution()),
337         (principal_ + interest_ + slippage_)
338     );
339 
340     // Trigger Revenue Distribution contract to distribute the payment
341     dataVault.revenueDistribution().distribute(
342         principal_,
343         interest_,
344         slippage_
345     );
346 }
```

As a result, if the `paymentSource_` address still has the amount of tokens approved, the attacker can use the `repay()` function to drain the tokens from the `paymentSource_` address in order to repay any loan.

5.1.2. Remediation

Inspex suggests removing the `paymentSource_` parameter from all related functions and replacing it with `msg.sender` instead to directly transfer the token from the caller.

5.2. Loan Interest Miscalculation

ID	IDX-002
Target	MasterLiquidator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** High **Impact:** High Due to the double counting of `interestLocker`, the bad debt for the liquidation loan will be larger than usual. **Likelihood:** Medium This will only occur during the liquidation process when the `interestLocker` is not empty.
Status	Resolved The MoneySwitch team has resolved this issue since the `repay()` function mechanism is modified, the interest of loan position is now calculated by using the leftover principle that removes the need for the `interestLocker` in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.2.1. Description

In the partial payment flow, when the borrower pays a portion of the principal, the interest that came from the portion of the principal that the borrower wants to repay will be stopped calculating and stored in the `interestLocker` variable for use in the interest calculation later, following lines 305 - 308.

MasterLender.sol

```
288 function _repayPartial(
289     address paymentSource_,
290     Loan memory loan_,
291     uint256 amount_,
292     uint256 interest_
293 ) private {
294     // Payment Variables
295     uint256 principalPayment_ = amount_;
296     uint256 interestPayment_ = 0;
297
298     // If the amount_ is larger than principal then it covers the full
currentPrincipal
299     // and it covers partial payment of interest
300     if (amount_ > loan_.currentPrincipal) {
```

```
        principalPayment_ = loan_.currentPrincipal;
        interestPayment_ = amount_ - loan_.currentPrincipal;
    }
    // Store outstanding interest, extract any paidup interest
    dataVault.loanRegistry().increaseInterestLocker(
        loan_.wallet,
        (interest_ - interestPayment_)
    );

    // Make Payment
    _makePayment(paymentSource_, principalPayment_, interestPayment_, 0);

    // Calibrate other contract according to repayment amount
    _calibrateRepayment(loan_, principalPayment_);

    // Emit Event
    emit Repaid(loan_.wallet, paymentSource_, amount_);
}
```

When a liquidated loan occurs because the borrower has not paid the debt within the expected duration, the admin calls the `liquidateLoan()` function in the `MasterLiquidator` contract. This function determines the amount the borrower will be needed to pay, where the amount is a sum of interest, principal and `interestLocker`.

However, the debt to be paid is calculated from `loan.currentPrincipal + loan.interestLocker + interest_`, due to the `interest_` came from the `_calculateInterest(loan) + loan.interestLocker` means the `loan.interestLocker` will be double added. It results in the amount to repay being larger than it should be. At lines 64 - 67.

MasterLiquidator.sol

```
54 function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
55     Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
56     require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");
57
58     dataVault.borrowerRegistry().setStatus(
59         loan.owner,
60         BorrowerStatus.Defaulted
61     );
62
63     dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
64     uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
65     uint256 totalOwed_ = loan.currentPrincipal +
66         loan.interestLocker +
67         interest_;
68
69     uint256 availableCollateral_ = dataVault
```

```
            .creditProtectionPool()
            .getBalance();

        if (availableCollateral_ >= totalOwed_) {
            dataVault.creditProtectionPool().reimbursePool(totalOwed_);
        } else {
            dataVault.creditProtectionPool().reimbursePool(
                availableCollateral_
            );
            (
                uint256 mainImpairment_,
                uint256 feederImpairment_
            ) = _impairmentCalculator(
                    totalOwed_ - availableCollateral_,
                    loanId_
                );
            IMasterPool(loan.pool).impairInterestFactor(
                mainImpairment_,
                feederImpairment_
            );
        }
    }
```

5.2.2. Remediation

Inspex suggests removing the `loan.interestLocker` variable from the `totalOwed_` variable because the `loan.interestLocker` already added in the `interest_` variable at line 65.

MasterLiquidator.sol

```
54  function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
55      Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
56      require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");
57
58      dataVault.borrowerRegistry().setStatus(
59          loan.owner,
60          BorrowerStatus.Defaulted
61      );
62
63      dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
64      uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
65      uint256 totalOwed_ = loan.currentPrincipal + interest_;
66
67      uint256 availableCollateral_ = dataVault
68          .creditProtectionPool()
69          .getBalance();
70
71      if (availableCollateral_ >= totalOwed_) {
```

```
        dataVault.creditProtectionPool().reimbursePool(totalOwed_);
    } else {
        dataVault.creditProtectionPool().reimbursePool(
            availableCollateral_
        );
        (
            uint256 mainImpairment_,
            uint256 feederImpairment_
        ) = _impairmentCalculator(
                totalOwed_ - availableCollateral_,
                loanId_
            );
        IMasterPool(loan.pool).impairInterestFactor(
            mainImpairment_,
            feederImpairment_
        );
    }
}
```

5.3. Improper Withdrawal Amount

ID	IDX-003
Target	FeederPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The `FeederPool` contract withdraws an excessive amount of funds from the `MasterPool` contract, resulting in an excess amount being stuck in the `FeederPool` contract and the `MasterPool` contract having insufficient funds for all users to withdraw. **Likelihood:** Medium This issue will occur when the `CreditProtectionPool` contract has insufficient funds to reimburse bad debts on the platform.
Status	Resolved The MoneySwitch team has resolved this issue by sending `interestOwed_` to calculate withdrawal principal in the MasterPool contract in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.3.1. Description

The `FeederPool` contract is a scaled profit and loss for lenders, allowing a lender to deposit their principal to provide liquidity for borrowers. Where the profit is gained from collecting interest from the borrower, a portion of the borrowed interest will be calculated as a percentage increase from the `_interestFactorFeeder` state, which is an essential factor in the lender's profit calculation.

In the case that a liquidated loan occurs, causing the `_interestFactorFeeder` value to be less than the lender's `_depositorInterestFactor` value, the lender who withdraws at this time will lose a portion of the principal that is calculated by the `_calculateInterestOwed()` function.

FeederPool.sol

```
function _calculateInterestOwed(uint256 amount_)
    internal
    view
    returns (uint256)
{
    return
        (amount_ *
            (_depositorInterestFactor[msg.sender] - _interestFactorFeeder) *
            _scalingFactor) / (_WAD * 100);
```

```
}
```

So, when the lender withdraws their principal, the `withdrawAll()` function in the `FeederPool` contract will withdraw the lender's principal from the `MasterPool` contract. Next, the `FeederPool` contract will calculate the interest owed from the lender's principal as the `interestOwed_` variable as line 186.

FeederPool.sol

```
163 function withdrawAll() external {
164     require(_activeDepositor[msg.sender], "FP:NON_ACTIVE_DEPOSITOR");
165 
166     _masterPool.updatePoolFromFeeder();
167     _getInterestFactorFeeder();
168     _getInterestFactorUnimpaired();
169 
170     if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
171         uint256 interest_ = _calculateInterest(
172             _principalDeposits[msg.sender]
173         );
174 
175         uint256 excessInterest_ = _calculateExcessInterest(
176             _principalDeposits[msg.sender]
177         );
178 
179         _withdraw(
180             _principalDeposits[msg.sender],
181             interest_,
182             excessInterest_,
183             0
184         );
185     } else {
186         uint256 interestOwed_ = _calculateInterestOwed(
187             _principalDeposits[msg.sender]
188         );
189 
190         require(
191             _principalDeposits[msg.sender] >= interestOwed_,
192             "FP:INTEREST_DUE"
193         );
194 
195         uint256 excessInterest_ = _calculateExcessInterest(
196             _principalDeposits[msg.sender]
197         );
198 
199         _withdraw(
200             _principalDeposits[msg.sender],
201             0,
```

```
            excessInterest_,
            interestOwed_
        );
    }

    _principalDeposits[msg.sender] = 0;
    _activeDepositor[msg.sender] = false;
    _depositorInterestFactor[msg.sender] = 0;
}

```

Finally, the lender's principal will be transferred to the lender by subtracting with the `interestOwed_` variable. After the operation is completed, the `interestOwed_` amount is still stored in the `FeederPool` contract at lines 238 - 241.

FeederPool.sol

```
219 function _withdraw(
220     uint256 principalWithdraw_,
221     uint256 interest_,
222     uint256 excessInterest_,
223     uint256 interestOwed_
224 ) internal {
225     _updateRewardFactorLocal();
226     _depositorRewardLocker[msg.sender] +=
227         (principalWithdraw_ *
228             (_rewardFactorLocal - _depositorRewardFactor[msg.sender])) /
229         _WAD;
230 
231     _principalDepositTotal -= principalWithdraw_;
232 
233     _masterPool.withdrawFeeder(
234         principalWithdraw_,
235         interest_,
236         excessInterest_
237     );
238     _liquidityAsset.transfer(
239         msg.sender,
240         principalWithdraw_ + interest_ - interestOwed_
241     );
242     emit Withdrawn(msg.sender, principalWithdraw_ + interest_);
243 }
```

However, the `FeederPool` contract does not have any operation that handles the remaining `interestOwed_`, causing the `interestOwed_` amount to be stuck in the `FeederPool` contract forever.

As a result, the `MasterPool` contract will have insufficient funds for all lenders to withdraw their funds due to the `FeederPool` contract withdrawing an excessive amount of funds and freezing them in the `FeederPool` contract.

5.3.2. Remediation

Inspex suggests withdrawing only the remaining lender's funds after the `interestOwed_` from the `MasterPool` contract has been calculated.

For example, the remaining borrower funds can be calculated inside the `MasterPool` contract by sending the `interestOwed_` parameter into the `withdrawFeeder()` function, as shown in lines 233 - 238.

FeederPool.sol

```
219 function _withdraw(
220     uint256 principalWithdraw_,
221     uint256 interest_,
222     uint256 excessInterest_,
223     uint256 interestOwed_
224 ) internal {
225     _updateRewardFactorLocal();
226     _depositorRewardLocker[msg.sender] +=
227         (principalWithdraw_ *
228             (_rewardFactorLocal - _depositorRewardFactor[msg.sender])) /
229         _WAD;
230 
231     _principalDepositTotal -= principalWithdraw_;
232 
233     _masterPool.withdrawFeeder(
234         principalWithdraw_,
235         interest_,
236         excessInterest_,
237         interestOwed_
238     );
239     _liquidityAsset.transfer(
240         msg.sender,
241         principalWithdraw_ + interest_ - interestOwed_
242     );
243     emit Withdrawn(msg.sender, principalWithdraw_ + interest_);
244 }
```

After that, the `MasterPool` contract will transfer the exact amount of the borrower principal that was calculated to the `FeederPool` contract following lines 339 and 355.

MasterPool.sol

```
335 function withdrawFeeder(
336     uint256 principalAmount_,
337     uint256 interestAmount_,
338     uint256 excessAmount_,
339     uint256 interestOwed_
340 ) external {
341     require(
342         dataVault.poolRegistry().isValidPool(msg.sender),
343         "MP:NOT_APPROVED_POOL"
344     );
345     require(
346         principalAmount_ <= _principalDeposits[msg.sender],
347         "MP:NOT_ENOUGH_FUNDS"
348     );
349 
350     _principalDeposits[msg.sender] -= principalAmount_;
351     _principalDepositAllPool -= principalAmount_;
352 
353     IERC20(_liquidityAsset).transfer(
354         msg.sender,
355         principalAmount_ + interestAmount_ - interestOwed_
356     );
357 
358     IERC20(_liquidityAsset).transfer(
359         address(dataVault.creditProtectionPool()),
360         excessAmount_
361     );
362 
363     emit FeederWithdrawn(
364         msg.sender,
365         principalAmount_ + interestAmount_,
366         excessAmount_
367     );
368 }
```

5.4. Signature Reutilization

ID	IDX-004
Target	MasterLender
Category	Advanced Smart Contract Vulnerability
CWE	CWE-347: Improper Verification of Cryptographic Signature
Risk	**Severity:** Medium **Impact:** Medium The borrower's signature can be reused, resulting in anyone being able to issue a loan for this borrower until they reach the credit borrow limit. **Likelihood:** Medium This issue occurs when the borrower has issued a loan once, then anyone can issue a loan to them with their signature if credit is sufficient.
Status	Resolved The MoneySwitch team has resolved this issue by including a non-reusable nonce number to signature hash signing in order to prevent reuse of the signature hash in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.4.1. Description

In order to issue a loan, the borrower has to create a signature with the amount, duration, and nonce. Then the `borrow()` function will be called to issue the loan to the borrower. However, no check was implemented to prevent signature replay after using the `ECDSA.recover()` function to get the borrower address from the signature in line 69.

MasterLender.sol

```
60 function borrow(
61     uint256 amount_,
62     uint256 duration_,
63     uint256 nonce_,
64     bytes memory signature_
65 ) external isValidDuration(duration_) {
66     bytes32 dataHash = ECDSA.toEthSignedMessageHash(
67         keccak256(abi.encodePacked(amount_, duration_, nonce_))
68     );
69     address borrower = ECDSA.recover(dataHash, signature_);
70 
71     _isValidBorrower(borrower, amount_);
72 
73     address wallet = _createLoan(borrower, amount_, duration_);
```

```
    dataVault.borrowerRegistry().increaseTotalOutstanding(
        borrower,
        _toWad(amount_)
    );

    uint256 interestRate_ = dataVault
        .loanRegistry()
        .getLoan(wallet)
        .interestRate;

    address receiveAddress = dataVault
        .borrowerRegistry()
        .getBorrower(borrower)
        .receiveAddress;

    _pool.borrow(borrower, receiveAddress, amount_, interestRate_);

    emit Borrowed(borrower, wallet, receiveAddress, amount_, duration_);
}
```

The attacker can reuse the same signature by calling the `borrow()` function with all the same parameters to arbitrarily create a new loan for the borrower while the borrower's credit is sufficient.

5.4.2. Remediation

Inspex suggests implementing a signature replay prevention mechanism to ensure that the loan with this signature was already created and cannot be reused. For example adding a `nonce` variable to `Borrower` struct for storing the nonce for each borrower.

Borrower.sol

```
struct Borrower {
    address owner;
    address receiveAddress;
    uint256 creditLimit;
    uint256 totalOutstanding;
    BorrowerStatus status;
    uint256 nonce;
}
```

Create the `checkAndIncreaseNonce()` function for checking and updating nonce in the `BorrowerRegistry` contract.

BorrowerRegistry.sol

```
event CheckAndIncreaseNonce(address indexed borrower_, uint256 nonce_);
function checkAndIncreaseNonce(address borrower_, uint256 nonce_)
```

```
    external
    onlyApprovedContract
    onlyExistBorrower(borrower_)
{
    require(_borrowers[borrower_].nonce == nonce_,
"BR:BORROWER_NONCE_INVALID");
    _borrowers[borrower_].nonce++;

    emit CheckAndIncreaseNonce(
        borrower_,
        nonce_
    );
}
```

Call the `checkAndIncreaseNonce()` function in the `borrow()` function at line 71 to prevent signature replay attack.

MasterLender.sol

```
60 function borrow(
61     uint256 amount_,
62     uint256 duration_,
63     uint256 nonce_,
64     bytes memory signature_
65 ) external isValidDuration(duration_) {
66     bytes32 dataHash = ECDSA.toEthSignedMessageHash(
67         keccak256(abi.encodePacked(amount_, duration_, nonce_))
68     );
69     address borrower = ECDSA.recover(dataHash, signature_);
70 
71     dataVault.borrowerRegistry().checkAndIncreaseNonce(
72         borrower,
73         nonce_
74     );
75     _isValidBorrower(borrower, amount_);
76 
77     address wallet = _createLoan(borrower, amount_, duration_);
78 
79     dataVault.borrowerRegistry().increaseTotalOutstanding(
80         borrower,
81         _toWad(amount_)
82     );
83 
84     uint256 interestRate_ = dataVault
85         .loanRegistry()
86         .getLoan(wallet)
87         .interestRate;
88 
```

```
    address receiveAddress = dataVault
        .borrowerRegistry()
        .getBorrower(borrower)
        .receiveAddress;

    _pool.borrow(borrower, receiveAddress, amount_, interestRate_);

    emit Borrowed(borrower, wallet, receiveAddress, amount_, duration_);
}
```

5.5. Centralized Control of State Variable

ID	IDX-005
Target	Governance InterestRateCalculator
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved The MoneySwitch team has resolved this issue as suggested by using the `onlyAllGovernance` modifier to control the use of critical state variables in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.5.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Contract	Function	Modifier
Governance.sol (L:113)	Governance	setPendingAdmin()	isAdmin
InterestRateCalculator.sol (L:39)	InterestRateCalculator	setBaseRate()	onlyAdmin
InterestRateCalculator.sol (L:47)	InterestRateCalculator	setCapRate()	onlyAdmin
InterestRateCalculator.sol (L:55)	InterestRateCalculator	setCapDuration()	onlyAdmin

InterestRateCalculator.sol (L:73)	InterestRateCalculator	setMaxLoanDuration()	onlyAdmin

5.5.2. Remediation

In the ideal case, the critical state variables should not be modifiable to maintain the integrity of the smart contract. However, if modifications are needed, Inspex suggests using the `onlyAllGovernance` modifier, which requires the governance decision instead of `onlyAdmin` modifier to control the use of these functions.

If removing the functions or using the `onlyAllGovernance` modifier is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours.

Please note that if the timelock mitigation is used for an admin role, the `suspendLoan()` and `liquidateLoan()` functions are also affected by the timelock, Inspex suggests implementing another modifier for these functions such as `onlyLiquidator` to avoid the timelock mechanism.

5.6. Insufficient Check for Approved Operation

ID	IDX-006
Target	Multiownable
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The admin or governor can execute the operation that was already approved without the approval of another party. **Likelihood:** Medium In order to perform this issue, the operation must be completely executed by both the admin and governor before. However, only the admin or governor can execute the operation again.
Status	Resolved The MoneySwitch team has resolved this issue by resetting operation approval to `false` after an operation is executed in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.6.1. Description

Due to a dual control design flaw, the `Multiownable` contract's `onlyAllGovernance()` modifier at line 14 requires both admin and governor approval before the privilege operation will be executed.

Multiownable.sol

```
14 modifier onlyAllGovernance() {
15     require(governance().isGovernance(msg.sender), "MOC:ONLY_GOVERNANCE");
16     bytes32 operation = _storeOperation();
17 
18     if (_isOperationApproved(operation)) {
19         _;
20     }
21 }
```

However, since the operation has already been set to true and will never be set back to false, when the admin and governor have both approved it, the admin or governor can immediately re-execute the operation without the approval of another party.

Multiownable.sol

```
function _isOperationApproved(bytes32 operation)
    private
    view
    returns (bool)
{
    return
        _operations[operation][governance().admin()] &&
        _operations[operation][governance().governor()];
}
```

For example, add the old contract that was already deleted by the `addContract()` function.

Governance.sol

```
function addContract(address key_) external onlyAllGovernance {
    _approvedContracts[key_] = true;
    _approvedContractKeys[_approvedContractCount] = key_;
    _approvedContractCount++;
}
```

Nevertheless, this issue is valid until the generation state is changed because the operation key is calculated from hashing the `msg.data` and governance's generation state.

Multiownable.sol

```
function _storeOperation() private returns (bytes32) {
    bytes32 operation = keccak256(
        abi.encodePacked(msg.data, governance().generation())
    );

    _operations[operation][msg.sender] = true;

    return operation;
}
```

5.6.2. Remediation

Inspex suggests updating the operation approval to false for both the admin and governor after the operation is completely executed.

For example, in the `onlyAllGovernance` modifier at lines 21 - 22.

Multiownable.sol

```
14 modifier onlyAllGovernance() {
15     require(governance().isGovernance(msg.sender), "MOC:ONLY_GOVERNANCE");
16     bytes32 operation = _storeOperation();
17 
18     if (_isOperationApproved(operation)) {
19         _;
20 
21         _operations[operation][governance().admin()] = false;
22         _operations[operation][governance().governor()] = false;
23     }
24 }
```

5.7. Improper Inactive Pools Handling

ID	IDX-007
Target	FeederPool MasterPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Medium **Impact:** Medium The user will be unable to withdraw their funds after the Feeder pool is inactive. Furthermore, if the platform has bad debt and the pool is impaired, the inactive Feeder pool is still affected. **Likelihood:** Medium This issue will occur when there are deposited funds left in the inactive feeder pool.
Status	Resolved The MoneySwitch team has resolved this issue by allowing users to withdraw funds from inactive pools in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.7.1. Description

The `withdrawInterestPrincipal()` and `withdrawAll()` functions in the `FeederPool` contract both call the `updatePoolFromFeeder()` function of the `MasterPool` contract at lines 118, 166.

FeederPool.sol

```
114 function withdrawInterestPrincipal(uint256 amount_) external {
115     require(amount_ > 0, "FP: NON_ZERO_INTEGER_ONLY");
116     require(_activeDepositor[msg.sender], "FP: NON_ACTIVE_DEPOSITOR");
117
118     _masterPool.updatePoolFromFeeder();
119
120     _getInterestFactorFeeder();
121     _getInterestFactorUnimpaired();
122
123     if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
124         uint256 principalWithdraw_ = _calculatePrincipalWithdraw(amount_);
125
126         require(
127             principalWithdraw_ <= _principalDeposits[msg.sender],
128             "FP: NOT_ENOUGH_FUNDS"
129         );
130
```

```
        uint256 interest_ = _calculateInterest(principalWithdraw_);

        uint256 excessInterest_ = _calculateExcessInterest(
            principalWithdraw_
        );

        _withdraw(principalWithdraw_, interest_, excessInterest_, 0);

        _principalDeposits[msg.sender] -= principalWithdraw_;
    } else {
        uint256 interestOwed_ = _calculateInterestOwed(
            _principalDeposits[msg.sender]
        );

        uint256 excessInterest_ = _calculateExcessInterest(amount_);

        require(amount_ >= interestOwed_, "FP:INTEREST_DUE");

        _depositorInterestFactor[msg.sender] = _interestFactorFeeder;

        _withdraw(amount_, 0, excessInterest_, interestOwed_);

        _principalDeposits[msg.sender] -= amount_;
    }
    if (_principalDeposits[msg.sender] == 0) {
        _activeDepositor[msg.sender] = false;
    }
}
```

FeederPool.sol

```
function withdrawAll() external {
    require(_activeDepositor[msg.sender], "FP:NON_ACTIVE_DEPOSITOR");

    _masterPool.updatePoolFromFeeder();
    _getInterestFactorFeeder();
    _getInterestFactorUnimpaired();

    if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
        uint256 interest_ = _calculateInterest(
            _principalDeposits[msg.sender]
        );

        uint256 excessInterest_ = _calculateExcessInterest(
            _principalDeposits[msg.sender]
        );

        _withdraw(
```

```
            _principalDeposits[msg.sender],
            interest_,
            excessInterest_,
            0
        );
    } else {
        uint256 interestOwed_ = _calculateInterestOwed(
            _principalDeposits[msg.sender]
        );

        require(
            _principalDeposits[msg.sender] >= interestOwed_,
            "FP:INTEREST_DUE"
        );

        uint256 excessInterest_ = _calculateExcessInterest(
            _principalDeposits[msg.sender]
        );

        _withdraw(
            _principalDeposits[msg.sender],
            0,
            excessInterest_,
            interestOwed_
        );
    }

    _principalDeposits[msg.sender] = 0;
    _activeDepositor[msg.sender] = false;
    _depositorInterestFactor[msg.sender] = 0;
}
```

The Feeder pool is required to be active when calling the `updatePoolFromFeeder()` function as shown in line 86. Thus, the withdrawal attempt afterward will always fail.

MasterPool.sol

```
85 function updatePoolFromFeeder() external {
86     require(
87         dataVault.poolRegistry().isActivePool(msg.sender),
88         "MP:NOT_APPROVED_POOL"
89     );
90
91     _updateCurrentTimeStamp();
92     _updateInterestFactor();
93     _updateLastTimeStamp();
94
95     emit Updated();
```

```
}
```

Furthermore, the `withdrawInterestPrincipal()` and `withdrawAll()` functions also call the `_getInterestFactorFeeder()` and `_getInterestFactorUnimpaired()` functions at lines 120 - 121, 167 - 168. which is normally used for updating the `_interestFactorFeeder` state to sync with the `MasterPool._interestFactorFeeder` and `MasterPool._interestFactorUnimpaired` states.

FeederPool.sol

```
114 function withdrawInterestPrincipal(uint256 amount_) external {
115     require(amount_ > 0, "FP: NON_ZERO_INTEGER_ONLY");
116     require(_activeDepositor[msg.sender], "FP: NON_ACTIVE_DEPOSITOR");
117 
118     _masterPool.updatePoolFromFeeder();
119 
120     _getInterestFactorFeeder();
121     _getInterestFactorUnimpaired();
122 
123     if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
124         uint256 principalWithdraw_ = _calculatePrincipalWithdraw(amount_);
```

FeederPool.sol

```
163 function withdrawAll() external {
164     require(_activeDepositor[msg.sender], "FP:NON_ACTIVE_DEPOSITOR");
165 
166     _masterPool.updatePoolFromFeeder();
167     _getInterestFactorFeeder();
168     _getInterestFactorUnimpaired();
169 
170     if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
171         uint256 interest_ = _calculateInterest(
172             _principalDeposits[msg.sender]
173         );
```

However, with the current design the `_interestFactorFeeder` and `_interestFactorUnimpaired` states are still updated after the pool is inactive. Resulting in the deposited funds in the inactive pool still affected by any interest or impairment of the platform.

5.7.2. Remediation

Inspex suggests skipping calls to the `updatePoolFromFeeder()`, `_getInterestFactorFeeder()`, and `_getInterestFactorUnimpaired()` functions when the Feeder pool is inactive to handle the deposited funds left in the Feeder pool.

For example adding the condition check whether the pool is inactive before calling these functions:

FeederPool.sol

```
function withdrawInterestPrincipal(uint256 amount_) external {
    require(amount_ > 0, "FP: NON_ZERO_INTEGER_ONLY");
    require(_activeDepositor[msg.sender], "FP: NON_ACTIVE_DEPOSITOR");

    if (dataVault.poolRegistry().isActivePool(address(this))) {
        _masterPool.updatePoolFromFeeder();
        _getInterestFactorFeeder();
        _getInterestFactorUnimpaired();
    }

    if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
        uint256 principalWithdraw_ = _calculatePrincipalWithdraw(amount_);
```

FeederPool.sol

```
function withdrawAll() external {
    require(_activeDepositor[msg.sender], "FP:NON_ACTIVE_DEPOSITOR");

    if (dataVault.poolRegistry().isActivePool(address(this))) {
        _masterPool.updatePoolFromFeeder();
        _getInterestFactorFeeder();
        _getInterestFactorUnimpaired();
    }

    if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
        uint256 interest_ = _calculateInterest(
            _principalDeposits[msg.sender]
        );
```

5.8. Reentrancy Attack

ID	IDX-008
Target	FeederPool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Medium **Impact:** High The attacker can drain tokens from the platform by abusing the `withdrawAll()` function of the `FeederPool` contract with the reentrancy attack. **Likelihood:** Low The reentrancy attack will occur when there is a callback while transferring the asset token, such as an ERC777 token. However, only the platform owner can set the liquidity asset token of the platform.
Status	Resolved The MoneySwitch team has resolved this issue as suggested by adding the "Checks-Effects-Interactions" pattern in the `_withdraw()` function and mutex-lock in the `withdrawAll()` function in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.8.1. Description

The `withdrawAll()` function in the `FeederPool` contract calls the `_withdraw()` function before setting the `_principalDeposits[msg.sender]` state to 0 in lines 199 and 207.

FeederPool.sol

```
163 function withdrawAll() external {
164     require(_activeDepositor[msg.sender], "FP:NON_ACTIVE_DEPOSITOR");
165
166     _masterPool.updatePoolFromFeeder();
167     _getInterestFactorFeeder();
168     _getInterestFactorUnimpaired();
169
170     if (_interestFactorFeeder >= _depositorInterestFactor[msg.sender]) {
171         uint256 interest_ = _calculateInterest(
172             _principalDeposits[msg.sender]
173         );
174
175         uint256 excessInterest_ = _calculateExcessInterest(
176             _principalDeposits[msg.sender]
```

```
        );

        _withdraw(
            _principalDeposits[msg.sender],
            interest_,
            excessInterest_,
            0
        );
    } else {
        uint256 interestOwed_ = _calculateInterestOwed(
            _principalDeposits[msg.sender]
        );

        require(
            _principalDeposits[msg.sender] >= interestOwed_,
            "FP:INTEREST_DUE"
        );

        uint256 excessInterest_ = _calculateExcessInterest(
            _principalDeposits[msg.sender]
        );

        _withdraw(
            _principalDeposits[msg.sender],
            0,
            excessInterest_,
            interestOwed_
        );
    }

    _principalDeposits[msg.sender] = 0;
    _activeDepositor[msg.sender] = false;
    _depositorInterestFactor[msg.sender] = 0;
}
```

The `_withdraw()` function has an external call to `ERC20.transfer()` function at line 238.

FeederPool.sol

```
219 function _withdraw(
220     uint256 principalWithdraw_,
221     uint256 interest_,
222     uint256 excessInterest_,
223     uint256 interestOwed_
224 ) internal {
225     _updateRewardFactorLocal();
226     _depositorRewardLocker[msg.sender] +=
227         (principalWithdraw_ *
```

```
            (_rewardFactorLocal - _depositorRewardFactor[msg.sender])) /
        _WAD;

    _principalDepositTotal -= principalWithdraw_;

    _masterPool.withdrawFeeder(
        principalWithdraw_,
        interest_,
        excessInterest_
    );
    _liquidityAsset.transfer(
        msg.sender,
        principalWithdraw_ + interest_ - interestOwed_
    );
    emit Withdrawn(msg.sender, principalWithdraw_ + interest_);
}
```

Normally the `ERC20.transfer()` function does not contain any hook or callback. However, if the platform owner sets the token which has a hook or callback while transferring tokens such as ERC777 (ERC20 compatible with send/receive hooks), the `withdrawAll()` function will be vulnerable to the reentrancy attack.

The attacker can use this flaw to perform the token drain from the `MasterPool` contract with the following example scenario:

1. Attacker creates a malicious contract which has a hook to call the `withdrawAll()` function on receiving tokens.
2. Malicious contract calls the `deposit()` function to deposit 100 tokens to the Feeder pool.
3. Malicious contract calls the `withdrawAll()` function to withdraw funds.
4. Upon 100 tokens are transferred back the malicious contract hook will call to the `withdrawAll()` again before the `_principalDeposits[msg.sender]` state is set to 0.
5. The 100 tokens will be withdrawn repeatedly until all tokens are drained from the pool.
6. Then the `_principalDeposits[msg.sender]` state is set to 0.

5.8.2. Remediation

Inspex suggests implementing the `withdrawAll()` function with the "Checks-Effects-Interactions" pattern to completely update the states before invoking external accounts or contracts. Or implement a mutex lock to prevent a reentrant calling to the same contract such as using the OpenZeppelin's `ReentrancyGuard` contract and adding the `nonReentrant` modifier to the `withdrawAll()` function.

5.9. Borrowing Credit Interest Not Included

ID	IDX-009
Target	LoanRegistry MasterLender
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Medium **Impact:** Medium Users can borrow the tokens more than the credit limit given by the platform. **Likelihood:** Medium It is very likely to occur when the users partially repay the loan to cover the principal tokens.
Status	Resolved The MoneySwitch team has resolved this issue since the modifying of the `repay()` function mechanism, the interest of loan position is now calculated by using the leftover principle that removes the need for the `interestLocker` in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.9.1. Description

In `MasterLender` contract, users can partially repay the loan by calling the `repay()` function and pass the `amount_` parameter less than `loan_.currentPrincipal + interestRounded_` as shown below in line 149:

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
127 ) external {
128     // Get Loan by ID
129     Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);
130 
131     // Check if loan is active
132     require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");
133 
134     // Get current timestamp
135     uint256 ts_ = block.timestamp;
136 
137     // Get total outstanding interestRounded
```

```
    uint256 interestRounded_ = _calculateInterestRounded(
        loan_,
        loan_.currentPrincipal,
        ts_
    ) + loan_.interestLocker;

    // Get interest on repayment principal
    uint256 interest_ = _calculateInterest(loan_, amount_, ts_);

    // If the amt is smaller than than current principal + interestRounded then it is
    // partial payment, otherwise it is a full payment
    if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
        // Make partial principal repayment
        _repayPartial(paymentSource_, loan_, amount_, interest_);
    } else {
        // Make full payment
        _repayFull(paymentSource_, loan_, interestRounded_, ts_);
    }
}
```

After calling the `_repayPartial()` function, if the value of the `amount_` is more than the principal, the interest will be leftover and added to the `interestLocker` in line 305 then the `totalOutstanding` and `currentPrincipal` will be decreased by the repayment amount in line 355 - 361 as shown below:

MasterLender.sol

```
288 function _repayPartial(
289     address paymentSource_,
290     Loan memory loan_,
291     uint256 amount_,
292     uint256 interest_
293 ) private {
294     // Payment Variables
295     uint256 principalPayment_ = amount_;
296     uint256 interestPayment_ = 0;
297
298     // If the amount_ is larger than principal then it covers the full currentPrincipal
299     // and it covers partial payment of interest
300     if (amount_ > loan_.currentPrincipal) {
301         principalPayment_ = loan_.currentPrincipal;
302         interestPayment_ = amount_ - loan_.currentPrincipal;
303     }
304     // Store outstanding interest, extract any paidup interest
305     dataVault.loanRegistry().increaseInterestLocker(
306         loan_.wallet,
```

```
307             (interest_ - interestPayment_)
308         );
309 
310         // Make Payment
311         _makePayment(paymentSource_, principalPayment_, interestPayment_, 0);
312 
313         // Calibrate other contract according to repayment amount
314         _calibrateRepayment(loan_, principalPayment_);
315 
316         // Emit Event
317         emit Repaid(loan_.wallet, paymentSource_, amount_);
318     }
```

MasterLender.sol

```
353 function _calibrateRepayment(Loan memory loan_, uint256 amount_) private {
354     _pool.repay(amount_, loan_.interestRate);
355     dataVault.loanRegistry().decreaseCurrentPrincipal(
356         loan_.wallet,
357         amount_
358     );
359     dataVault.borrowerRegistry().decreaseTotalOutstanding(
360         loan_.owner,
361         _toWad(amount_)
362     );
363 }
```

From the process above, after the `totalOutstanding` and `currentPrincipal` is decreased, the user can call the `borrow()` function to borrow tokens more than it should be since the validation check at line 426 in the `_isValidBorrower()` function does not count the leftover interest as the debt.

MasterLender.sol

```
60 function borrow(
61     uint256 amount_,
62     uint256 duration_,
63     uint256 nonce_,
64     bytes memory signature_
65 ) external isValidDuration(duration_) {
66     bytes32 dataHash = ECDSA.toEthSignedMessageHash(
67         keccak256(abi.encodePacked(amount_, duration_, nonce_))
68     );
69     address borrower = ECDSA.recover(dataHash, signature_);
70 
71     _isValidBorrower(borrower, amount_);
72 
73     address wallet = _createLoan(borrower, amount_, duration_);
74 
```

```
    dataVault.borrowerRegistry().increaseTotalOutstanding(
        borrower,
        _toWad(amount_)
    );

    uint256 interestRate_ = dataVault
        .loanRegistry()
        .getLoan(wallet)
        .interestRate;

    address receiveAddress = dataVault
        .borrowerRegistry()
        .getBorrower(borrower)
        .receiveAddress;

    _pool.borrow(borrower, receiveAddress, amount_, interestRate_);

    emit Borrowed(borrower, wallet, receiveAddress, amount_, duration_);
}
```

MasterLender.sol

```
function _isValidBorrower(address borrower_, uint256 amount_) private view {
    Borrower memory borrower = dataVault.borrowerRegistry().getBorrower(
        borrower_
    );
    require(
        borrower.status == BorrowerStatus.Active,
        "ML:NOT_ACTIVE_BORROWER"
    );
    require(
        borrower.creditLimit > borrower.totalOutstanding + _toWad(amount_),
        "ML:CREDIT_LIMIT_EXCEEDED"
    );
}
```

5.9.2. Remediation

Inspex suggests modifying the `increaseInterestLocker()` and `resetInterestLocker()` functions in `LoanRegistry` contract for adding leftover debt to the `borrower.totalOutstanding` as shown in line 112 and 128, for example:

LoanRegistry.sol

```
106 function increaseInterestLocker(address loanId_, uint256 interest_)
107     external
108     onlyApprovedContract
109     onlyExistLoan(loanId_)
```

```
{
    _loans[loanId_].interestLocker += interest_;
    dataVault.borrowerRegistry().increaseTotalOutstanding(
        _loans[loanId_].owner,
        _loans[loanId_].interestLocker
    );
    emit InterestLockerIncreased(loanId_, interest_);
}

/**
    @dev    Resets interest locker of a given loan.
    @param  loanId_ Address of loan.
 */
function resetInterestLocker(address loanId_)
    external
    onlyApprovedContract
    onlyExistLoan(loanId_)
{
    dataVault.borrowerRegistry().decreaseTotalOutstanding(
        _loans[loanId_].owner,
        _loans[loanId_].interestLocker
    );
    _loans[loanId_].interestLocker = 0;

    emit InterestLockerReseted(loanId_);
}
```

5.10. Loan Repayment Date Not Enforced

ID	IDX-010
Target	InterestRateCalculator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Medium **Impact:** Low The platform and lender benefit will be less than expected due to the low borrow interest rate. **Likelihood:** High It is very likely to occur when borrowing since there is no cost for performing this attack scenario and the attacker will pay less interest when repaying the debt.
Status	Resolved * The MoneySwitch team has clarified that providing uncollateralized lending is important to MoneySwitch's customers. There are controls over the borrower to protect the users, as follows: - All borrowers are white-listed and have restricted credit limits. The borrower has to be onboarded with licensed cross-border payment companies; these licenses are worth between $5 - $10 million USD, depending on the jurisdiction; thus, instead of taking collateral in the form of a volatile crypto asset, they take it in the form of a real-world contractual asset. - MoneySwitch's customers are multinational payment companies that operate in a heavily regulated industry and tend to be well capitalized. - The credit protection pool will increase overtime to mitigate this risk. - The MoneySwitch is transparent about the fact that the loans are uncollateralized; the interest rate is reflective of these risks.

5.10.1. Description

The MoneySwitch is a fixed-rate lending platform; `_baseRate` is currently fixed at 0.1% per day for up to 20 days. Alternatively, a borrower may borrow for up to 30 days, but the interest rate is 2% per borrowing duration, as shown in line 31.

InterestRateCalculator.sol

```
23 function calculateInterestRate(uint256 duration_)
24     external
25     view
```

```
    returns (uint256)
{
    if (duration_ <= _capDuration) {
        return _baseRate; // 0.1 per day in WAD
    } else {
        return (_capRate) / duration_; // rate per day in WAD (cap at _capRate)
    }
}
```

With the current design, the users can borrow with 30 days duration to get the lowest interest rate which is 0.067% (2%/30) and repay the loan before the duration expires. This results in causing less benefit for the platform and lenders.

5.10.2. Remediation

Inspex suggests adding a validation in `repay()` function at line 131 - 133, for example:

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
127 ) external {
128     // Get Loan by ID
129     Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);
130 
131     if(loan_.duration > dataVault.interestRateCalculator().capDuration()){
132         require(block.timestamp >= loan_.startDate + (loan_.duration * 1 days),
"MLI:LOAN_NOT_VALID_DATE");
133     }
134     // Check if loan is active
135     require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");
136 
137     // Get current timestamp
138     uint256 ts_ = block.timestamp;
139 
140     // Get total outstanding interestRounded
141     uint256 interestRounded_ = _calculateInterestRounded(
142         loan_,
143         loan_.currentPrincipal,
144         ts_
145     ) + loan_.interestLocker;
146 
147     // Get interest on repayment principal
148     uint256 interest_ = _calculateInterest(loan_, amount_, ts_);
149 
150     // If the amt is smaller than than current principal + interestRounded then
```

```
it is
    // partial payment, otherwise it is a full payment
    if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
        // Make partial principal repayment
        _repayPartial(paymentSource_, loan_, amount_, interest_);
    } else {
        // Make full payment
        _repayFull(paymentSource_, loan_, interestRounded_, ts_);
    }
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.11. Insufficient Liquidation Flow Check

ID	IDX-011
Target	MasterLiquidator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The admin can suspend the loan and change the borrower status without any criteria. Thus, it will make a bad debt to the platform which causes a damage to the principal of the lenders. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the admin.
Status	Resolved * The MoneySwitch team has clarified that the ability to suspend or liquidate the loan before the end of the loan duration is needed in several situations (e.g., regulatory, client requests, the default of another loan by the same borrower, required AML checks, and so forth). Retaining the flexibility to suspend a loan as early as possible in the default process is important to prevent interest from accruing that is unlikely to be paid. Furthermore, they have mitigated this issue by changing the liquidation of loans to be done through the governance instead of only admin in commit `48f7a872353c957809239676615e0920d3eb3b95` to provide better security against loan liquidation.

5.11.1. Description

In the `MasterLiquidator` contract, the admin can call the `suspendLoan()` function to suspend the loan.

However, the loan can be arbitrarily suspended before the loan duration expires since the `suspendLoan()` function has no validation to prevent this action as shown below.

MasterLiquidator.sol

```
function suspendLoan(address loanId_) external onlyAdmin(msg.sender) {
    Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
    require(loan.status == LoanStatus.Active, "MLI:LOAN_NOT_ACTIVE");

    dataVault.borrowerRegistry().setStatus(
        loan.owner,
```

```
        BorrowerStatus.Suspended
    );

    dataVault.loanRegistry().suspend(loanId_, block.timestamp);

    IMasterPool(loan.pool).suspend(
        loan.currentPrincipal,
        loan.interestRate
    );
}
```

Moreover, the suspended loan can be liquidated by calling the **liquidateLoan()** function without a validation check on the duration for repayment as shown below.

MasterLiquidator.sol

```
function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
    Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
    require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");

    dataVault.borrowerRegistry().setStatus(
        loan.owner,
        BorrowerStatus.Defaulted
    );

    dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
    uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
    uint256 totalOwed_ = loan.currentPrincipal +
        loan.interestLocker +
        interest_;

    uint256 availableCollateral_ = dataVault
        .creditProtectionPool()
        .getBalance();

    if (availableCollateral_ >= totalOwed_) {
        dataVault.creditProtectionPool().reimbursePool(totalOwed_);
    } else {
        dataVault.creditProtectionPool().reimbursePool(
            availableCollateral_
        );
        (
            uint256 mainImpairment_,
            uint256 feederImpairment_
        ) = _impairmentCalculator(
                totalOwed_ - availableCollateral_,
                loanId_
            );
```

```
        IMasterPool(loan.pool).impairInterestFactor(
            mainImpairment_,
            feederImpairment_
        );
    }
}
```

This results in causing a bad debt to the platform and is unfair to the lenders.

5.11.2. Remediation

Inspex suggests adding a validation in `suspendLoan()` and `liquidateLoan()` functions at lines 32 and 58, for example:

MasterLiquidator.sol

```
29 uint256 private repayDuration = 15 days;
30 function suspendLoan(address loanId_) external onlyAdmin(msg.sender) {
31     Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
32     require(block.timestamp > loan.startDate + (loan.duration * 1 days),
"MLI:LOAN_NOT_EXPIRED");
33     require(loan.status == LoanStatus.Active, "MLI:LOAN_NOT_ACTIVE");
34
35     dataVault.borrowerRegistry().setStatus(
36         loan.owner,
37         BorrowerStatus.Suspended
38     );
39
40     dataVault.loanRegistry().suspend(loanId_, block.timestamp);
41
42     IMasterPool(loan.pool).suspend(
43         loan.currentPrincipal,
44         loan.interestRate
45     );
46 }
47
48 /**********************************/
49 /*** Liquidation Functionality ***/
50 /**********************************/
51
52 /**
53     @dev    Liquidates loan, moving collateral, and updating factors
54     @param  loanId_ Wallet containing Loan to be suspended.
55 */
56 function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
57     Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
58     require(block.timestamp > loan.suspensionDate + repayDuration,
"MLI:LOAN_IN_DURATION");
```

```
    require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");

    dataVault.borrowerRegistry().setStatus(
        loan.owner,
        BorrowerStatus.Defaulted
    );

    dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
    uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
    uint256 totalOwed_ = loan.currentPrincipal +
        loan.interestLocker +
        interest_;

    uint256 availableCollateral_ = dataVault
        .creditProtectionPool()
        .getBalance();

    if (availableCollateral_ >= totalOwed_) {
        dataVault.creditProtectionPool().reimbursePool(totalOwed_);
    } else {
        dataVault.creditProtectionPool().reimbursePool(
            availableCollateral_
        );
        (
            uint256 mainImpairment_,
            uint256 feederImpairment_
        ) = _impairmentCalculator(
                totalOwed_ - availableCollateral_,
                loanId_
            );
        IMasterPool(loan.pool).impairInterestFactor(
            mainImpairment_,
            feederImpairment_
        );
    }
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.12. Business Design Flaw

ID	IDX-012
Target	MasterLiquidator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium When the borrower cannot repay the loan within the loan duration, the platform owner will liquidate the loan. However, since it is an uncollateralized loan, there is no collateral asset to be seized. Resulting in a bad debt to all platform users. **Likelihood:** Medium Bad debt only occurs when the loan was not fully repaid. Furthermore, the borrower has to be added to the platform by the Admin and Governance roles, which also assign the credit limit for each borrower. Therefore, it is unlikely that the whitelisted borrowers will not repay the debt.
Status	Resolved * The MoneySwitch team has clarified that providing uncollateralized lending is important to MoneySwitch's customers. There are controls over the borrower to protect the users, as follows: - All borrowers are white-listed and have restricted credit limits. The borrower has to be onboarded with licensed cross-border payment companies; these licenses are worth between $5 - $10 million USD, depending on the jurisdiction; thus, instead of taking collateral in the form of a volatile crypto asset, they take it in the form of a real-world contractual asset. - MoneySwitch's customers are multinational payment companies that operate in a heavily regulated industry and tend to be well capitalized. - The credit protection pool will increase overtime to mitigate this risk. - The MoneySwitch is transparent about the fact that the loans are uncollateralized; the interest rate is reflective of these risks.

5.12.1. Description

In the liquidation flow, after the borrower does not repay the loan within the duration, the admin role will call the `suspendLoan()` function in the `MasterLiquidator` contract.

MasterLiquidator.sol

```
function suspendLoan(address loanId_) external onlyAdmin(msg.sender) {
```

```
    Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
    require(loan.status == LoanStatus.Active, "MLI:LOAN_NOT_ACTIVE");

    dataVault.borrowerRegistry().setStatus(
        loan.owner,
        BorrowerStatus.Suspended
    );

    dataVault.loanRegistry().suspend(loanId_, block.timestamp);

    IMasterPool(loan.pool).suspend(
        loan.currentPrincipal,
        loan.interestRate
    );
}
```

After an off-chain debt collection process, if there is still a debt that has not been repaid, the loan will be liquidated by using the `liquidateLoan()` function. Then the `impairInterestFactor()` function will be called to set the impairment factor in the `MasterPool` contract at line 86.

MasterLiquidator.sol

```
function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
    Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
    require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");

    dataVault.borrowerRegistry().setStatus(
        loan.owner,
        BorrowerStatus.Defaulted
    );

    dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
    uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
    uint256 totalOwed_ = loan.currentPrincipal +
        loan.interestLocker +
        interest_;

    uint256 availableCollateral_ = dataVault
        .creditProtectionPool()
        .getBalance();

    if (availableCollateral_ >= totalOwed_) {
        dataVault.creditProtectionPool().reimbursePool(totalOwed_);
    } else {
        dataVault.creditProtectionPool().reimbursePool(
            availableCollateral_
        );
```

```
        (
            uint256 mainImpairment_,
            uint256 feederImpairment_
        ) = _impairmentCalculator(
                totalOwed_ - availableCollateral_,
                loanId_
            );
        IMasterPool(loan.pool).impairInterestFactor(
            mainImpairment_,
            feederImpairment_
        );
    }
}
```

The `impairInterestFactor()` function is used for reducing the `_interestFactorMain` and `_interestFactorFeeder` states at lines 437 - 438.

MasterPool.sol

```
429 function impairInterestFactor(
430     uint256 mainImpairment_,
431     uint256 feederImpairment_
432 ) external onlyApprovedContract {
433     require(
434         address(dataVault.masterLiquidator()) == msg.sender,
435         "MP:NOT_MASTER_LIQUIDATOR"
436     );
437     _interestFactorMain -= mainImpairment_;
438     _interestFactorFeeder -= feederImpairment_;
439 
440     emit InterestFactorImpaired(
441         msg.sender,
442         mainImpairment_,
443         feederImpairment_
444     );
445 }
```

The `_interestFactorMain` and `_interestFactorFeeder` states are commonly used for calculating the token amount per deposited principle for each user. If the factor is lower than the `_depositorInterestFactor[msg.sender]` state, it means the user can withdraw fewer tokens than the deposited amount as shown in line 274.

MasterPool.sol

```
252 function withdrawAll() external {
253     require(_activeDepositor[msg.sender], "MP:NON_ACTIVE_DEPOSITOR");
254 
```

```
        _updateCurrentTimeStamp();
        _updateInterestFactor();
        _updateLastTimeStamp();

        if (_interestFactorMain >= _depositorInterestFactor[msg.sender]) {
            uint256 amount_ = _principalDeposits[msg.sender] +
                _calculateInterest(_principalDeposits[msg.sender]);

            _withdraw(amount_, _principalDeposits[msg.sender]);
        } else {
            uint256 interestOwed_ = _calculateinterestOwed(
                _principalDeposits[msg.sender]
            );

            require(
                _principalDeposits[msg.sender] >= interestOwed_,
                "MP:INTEREST_DUE"
            );

            _withdraw(
                _principalDeposits[msg.sender] - interestOwed_,
                _principalDeposits[msg.sender]
            );
        }

        _principalDeposits[msg.sender] = 0;
        _activeDepositor[msg.sender] = false;
        _depositorInterestFactor[msg.sender] = 0;
    }
```

Therefore, with the current design, the advanced user can monitor the mempool for the suspend or liquidate event, then withdraw the funds before the loan is liquidated to avoid loss of funds.

5.12.2. Remediation

Inspex suggests redesigning the liquidation flow of the platform. For example, the borrower must have a locked collateral before creating a loan to ensure that the bad debt issue will not occur when the loan is liquidated.

5.13. Improper Feeder Pool Count Increment

ID	IDX-013
Target	PoolRegistry
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The `_feederPoolCount` state will be used in the `_impairmentCalculator()` function. This results in a miscalculation of the `_interestFactorMain` and `_interestFactorFeeder` states. **Likelihood:** Medium It is very likely that miscalculation of the `_interestFactorMain` and `_interestFactorFeeder` states will occur after executing the `setFeederPoolStatus()` function.
Status	Resolved The MoneySwitch team has resolved this issue as suggested by removing the changing state of the `_feederPoolKeys` and `_feederPoolCount` when execute the `setFeederPoolStatus()` function in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.13.1. Description

In the PoolRegistry contract, the `setFeederPoolStatus()` function is used for set status of the feeder pool.

However, changing the status of the pool should not change the state of `_feederPoolCount` that is the length of the pool in the `PoolRegistry` contract as shown below in lines 76 - 77:

PoolRegistry.sol

```
66 function setFeederPoolStatus(IFeederPool feederPool_, PoolStatus status_)
67     external
68     onlyAllGovernance
69 {
70     require(status_ != PoolStatus.Initialized, "PR:INVALID_STATUS");
71     require(
72         _feederPools[feederPool_] != PoolStatus.Initialized,
73         "PR:POOL_NOT_FOUND"
74     );
75     _feederPools[feederPool_] = status_;
76     _feederPoolKeys[_feederPoolCount] = feederPool_;
```

```
    _feederPoolCount++;

    emit FeederPoolStatusChanged(feederPool_, status_);
}
```

Furthermore, the improper state of `_feederPoolCount` will affect the value of `_interestFactorMain` and `_interestFactorFeeder` when being execute in the `IMasterPool(loan.pool).impairInterestFactor()` function at line 86.

It also affects the `_impairmentCalculator()` function at lines 79 - 85 that is calculated from the `_feederPoolCount` at lines 115 and 126 in the `MasterLiquidator` contract as shown below.

MasterLiquidator.sol

```
54 function liquidateLoan(address loanId_) external onlyAdmin(msg.sender) {
55     Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);
56     require(loan.status == LoanStatus.Suspended, "MLI:LOAN_NOT_SUSPENDED");
57 
58     dataVault.borrowerRegistry().setStatus(
59         loan.owner,
60         BorrowerStatus.Defaulted
61     );
62 
63     dataVault.loanRegistry().setStatus(loanId_, LoanStatus.Defaulted);
64     uint256 interest_ = _calculateInterest(loan) + loan.interestLocker;
65     uint256 totalOwed_ = loan.currentPrincipal +
66         loan.interestLocker +
67         interest_;
68 
69     uint256 availableCollateral_ = dataVault
70         .creditProtectionPool()
71         .getBalance();
72 
73     if (availableCollateral_ >= totalOwed_) {
74         dataVault.creditProtectionPool().reimbursePool(totalOwed_);
75     } else {
76         dataVault.creditProtectionPool().reimbursePool(
77             availableCollateral_
78         );
79         (
80             uint256 mainImpairment_,
81             uint256 feederImpairment_
82         ) = _impairmentCalculator(
83                 totalOwed_ - availableCollateral_,
84                 loanId_
85             );
86         IMasterPool(loan.pool).impairInterestFactor(
```

```
            mainImpairment_,
            feederImpairment_
        );
    }
}

/*********************************/
/*** Helper Functions ***/
/*********************************/

/**
    @dev    Calculates the amount to impair interest factors by.
    @param  amount_ Size of impairment.
    @param  loanId_ Wallet containing Loan to be impaired.
*/
function _impairmentCalculator(uint256 amount_, address loanId_)
    internal
    view
    returns (uint256, uint256)
{
    Loan memory loan = dataVault.loanRegistry().getLoan(loanId_);

    uint256 totalFeederSize_ = 0;
    uint256 mainSize_ = IMasterPool(loan.pool).principalDepositTotal();
    uint256 mainImpairment_ = (amount_ * _WAD) / mainSize_;

    for (
        uint256 i = 0;
        i < dataVault.poolRegistry().feederPoolCount();
        i++
    ) {
        IFeederPool feederPool = dataVault
            .poolRegistry()
            .getFeederPoolByIdx(i);
        totalFeederSize_ += feederPool.principalDepositTotal();
    }

    for (
        uint256 i = 0;
        i < dataVault.poolRegistry().feederPoolCount();
        i++
    ) {
        IFeederPool feederPool = dataVault
            .poolRegistry()
            .getFeederPoolByIdx(i);

        if (dataVault.poolRegistry().isActivePool(address(feederPool))) {
```

```
            mainImpairment_ -=
                (amount_ *
                    _WAD *
                    feederPool.principalDepositTotal() *
                    feederPool.scalingFactor()) /
                (mainSize_ * 100 * (mainSize_ + totalFeederSize_));
        }
    }
    uint256 feederImpairment_ = (amount_ * _WAD) /
        (mainSize_ + totalFeederSize_);
    return (mainImpairment_, feederImpairment_);
}
```

5.13.2. Remediation

Inspex suggests removing the changing state of the `_feederPoolKeys` and `_feederPoolCount`, for example:

PoolRegistry.sol

```
function setFeederPoolStatus(IFeederPool feederPool_, PoolStatus status_)
    external
    onlyAllGovernance
{
    require(status_ != PoolStatus.Initialized, "PR:INVALID_STATUS");
    require(
        _feederPools[feederPool_] != PoolStatus.Initialized,
        "PR:POOL_NOT_FOUND"
    );
    _feederPools[feederPool_] = status_;

    emit FeederPoolStatusChanged(feederPool_, status_);
}
```

5.14. Missing Input Validation

ID	IDX-014
Target	Governance
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The controlling authorities can set both admin and governor as the same address. This results in breaking the logic of dual signature requirements by admin and governor for critical contract operations. **Likelihood:** Low There is nothing to restrict the changes from being done; however, this action can only be done by the controlling authorities.
Status	Resolved The MoneySwitch team has resolved this issue by adding a validation check and using `_superAdmin` which is a multisig contract when set to `_governor` and `_admin` in commit `48f7a872353c957809239676615e0920d3eb3b95`. However, the multisig contract is not yet deployed during the reassessment, users should confirm that `_superAdmin` contract is a multisig contract.

5.14.1. Description

The logic of the dual signature requirement by admin and governor is used to execute the critical contract operations for the transparency of the platform.

However, the controlling authorities can set both admin and governor as the same address by the following methods.

- In the `Governance` contract, the contract owner can pass addresses of `admin_` and `governor_` as the same address when initialize the contract as shown below:

Governance.sol

```
constructor(address admin_, address governor_) {
    _admin = admin_;
    _governor = governor_;
}
```

- In `setPendingAdmin()` and `setPendingGovernor()` functions, the admin and governor can set _pendingAdmin and _pendingGovernor as the same address then execute acceptAdmin() and acceptGovernor() functions to become the admin and governor after that.

Governance.sol

```
113 function setPendingAdmin(address pendingAdmin_) external isAdmin {
114     require(pendingAdmin_ != address(0), "GOV:ZERO_ADDR");
115     require(pendingAdmin_ != _governor, "GOV:ALREADY_GOVERNOR");
116     _pendingAdmin = pendingAdmin_;
117 }
118
119 /**
120     @dev Allow proposed admin to accept the new role.
121 */
122 function acceptAdmin() external {
123     require(msg.sender == _pendingAdmin, "APP:NOT_PENDING_ADMIN");
124     _admin = msg.sender;
125     _pendingAdmin = address(0);
126     _generation++;
127 }
128
129 /**
130     @dev Propose a new governor address.
131 */
132 function setPendingGovernor(address pendingGovernor_) external isGovernor {
133     require(pendingGovernor_ != address(0), "APP:ZERO_ADDR");
134     require(pendingGovernor_ != _admin, "APP:ALREADY_ADMIN");
135     _pendingGovernor = pendingGovernor_;
136 }
137
138 /**
139     @dev Allow proposed governor to accept the new role.
140 */
141 function acceptGovernor() external {
142     require(_pendingGovernor == msg.sender, "APP:NOT_PENDING_GOV");
143     _governor = _pendingGovernor;
144     _pendingGovernor = address(0);
145     _generation++;
146 }
```

5.14.2. Remediation

Inspex suggests adding an input validation for admin_ and governor_ in constructor at line 47 and adding validation in setPendingAdmin() and setPendingGovernor() functions at line 116 and 135, for example:

Governance.sol

```
46 constructor(address admin_, address governor_) {
47     require(admin_ != governor_, "APP:SAME_ADDRESS");
```

```
    _admin = admin_;
    _governor = governor_;
}
```

Governance.sol

```
function setPendingAdmin(address pendingAdmin_) external isAdmin {
    require(pendingAdmin_ != address(0), "GOV:ZERO_ADDR");
    require(pendingAdmin_ != _governor, "GOV:ALREADY_GOVERNOR");
    require(pendingAdmin_ != _pendingGovernor, "GOV:ALREADY_PENDING_GOVERNOR");
    _pendingAdmin = pendingAdmin_;
}
```

Governance.sol

```
function setPendingGovernor(address pendingGovernor_) external isGovernor {
    require(pendingGovernor_ != address(0), "APP:ZERO_ADDR");
    require(pendingGovernor_ != _admin, "APP:ALREADY_ADMIN");
    require(pendingGovernor_ != _pendingAdmin, "GOV:ALREADY_PENDING_ADMIN");
    _pendingGovernor = pendingGovernor_;
}
```

5.15. Repay Interest Miscalculation

ID	IDX-015
Target	MasterLender
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Low **Impact:** Medium The borrowers who pay a partial debt, which is more than the principal but does not cover the interest, are more expensive than usual due to miscalculation. **Likelihood:** Low This will occur when the borrower accidentally enters an amount that is higher than the borrowed principal but lower than the total debt.
Status	Resolved The MoneySwitch team has resolved this issue by modifying the `repay()` function mechanism in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.15.1. Description

In the loan repayment flow, allowing a borrower to make a partial principal repayment by inputting the repay amount, in case that amount is greater than the borrowed principal, the excess amount will be paid as the borrowing interest.

Since the inputting amount is represented as a sum of borrowed principal and interest, calculating borrowing interest from the inputting amount is likely miscalculated in case the borrower repays their entire principal and a portion of borrowing interest. As a result, the borrowed interest is higher than it should be, as shown in line 145.

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
127 ) external {
128     // Get Loan by ID
129     Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);
130 
131     // Check if loan is active
132     require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");
133 
```

```
134     // Get current timestamp
135     uint256 ts_ = block.timestamp;
136
137     // Get total outstanding interestRounded
138     uint256 interestRounded_ = _calculateInterestRounded(
139         loan_,
140         loan_.currentPrincipal,
141         ts_
142     ) + loan_.interestLocker;
143
144     // Get interest on repayment principal
145     uint256 interest_ = _calculateInterest(loan_, amount_, ts_);
146
147     // If the amt is smaller than than current principal + interestRounded then it is
148     // partial payment, otherwise it is a full payment
149     if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
150         // Make partial principal repayment
151         _repayPartial(paymentSource_, loan_, amount_, interest_);
152     } else {
153         // Make full payment
154         _repayFull(paymentSource_, loan_, interestRounded_, ts_);
155     }
156 }
```

MasterLender.sol

```
376 function _calculateInterest(
377     Loan memory loan,
378     uint256 amount_,
379     uint256 ts_
380 ) private pure returns (uint256) {
381     return
382         (amount_ * (ts_ - loan.startDate) * loan.interestRate) /
383         (1 days * _WAD * 100);
384 }
```

5.15.2. Remediation

Inspex suggests adding a checking condition if the input amount exceeds the borrowed principal as an example shown in lines 145 - 150.

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
```

```
) external {
    // Get Loan by ID
    Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);

    // Check if loan is active
    require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");

    // Get current timestamp
    uint256 ts_ = block.timestamp;

    // Get total outstanding interestRounded
    uint256 interestRounded_ = _calculateInterestRounded(
        loan_,
        loan_.currentPrincipal,
        ts_
    ) + loan_.interestLocker;

    // Get interest on repayment principal
    uint256 interest_;
    if (amount_ >= loan_.currentPrincipal){
        interest_ = _calculateInterest(loan_, loan_.currentPrincipal, ts_);
    } else {
        interest_ = _calculateInterest(loan_, amount_, ts_);
    }

    // If the amt is smaller than than current principal + interestRounded then it is
    // partial payment, otherwise it is a full payment
    if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
        // Make partial principal repayment
        _repayPartial(paymentSource_, loan_, amount_, interest_);
    } else {
        // Make full payment
        _repayFull(paymentSource_, loan_, interestRounded_, ts_);
    }
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.16. Integer Underflow

ID	IDX-016
Target	MasterLender
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The `repay()` function in the `MasterLender` contract will be reverted. However, users can adjust the amount of repayment to avoid the revert. **Likelihood:** Low It is very unlikely for this issue to occur since the amount of repayment is input from users.
Status	Resolved The MoneySwitch team has resolved this issue by modifying the `repay()` function mechanism in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.16.1. Description

The `repay()` function in the `MasterLender` contract allows the user to repay a loan which can be partially repaid or fully repaid depending on the `amount` parameter.

In line 151, the `_repayPartial()` function will be called if the amount of payment is less than the combined of `loan_.currentPrincipal` and `interestRounded_`, but the variable that passed to the `_repayPartial()` is the `interest_` instead of `interestRounded_` as shown below:

MasterLender.sol

```
123 function repay(
124     address paymentSource_,
125     address loanId_,
126     uint256 amount_
127 ) external {
128     // Get Loan by ID
129     Loan memory loan_ = dataVault.loanRegistry().getLoan(loanId_);
130 
131     // Check if loan is active
132     require(loan_.status == LoanStatus.Active, "ML:LOAN_NOT_ACTIVE");
133 
134     // Get current timestamp
135     uint256 ts_ = block.timestamp;
136 
```

```
    // Get total outstanding interestRounded
    uint256 interestRounded_ = _calculateInterestRounded(
        loan_,
        loan_.currentPrincipal,
        ts_
    ) + loan_.interestLocker;

    // Get interest on repayment principal
    uint256 interest_ = _calculateInterest(loan_, amount_, ts_);

    // If the amt is smaller than than current principal + interestRounded then
it is
    // partial payment, otherwise it is a full payment
    if (amount_ < (loan_.currentPrincipal + interestRounded_)) {
        // Make partial principal repayment
        _repayPartial(paymentSource_, loan_, amount_, interest_);
    } else {
        // Make full payment
        _repayFull(paymentSource_, loan_, interestRounded_, ts_);
    }
}
```

This results in the `_repayPartial` function can be reverted at line 307 since the `interest_` can be less than the `interestPayment_` if the value of

`loan_.currentPrincipal + interest_` < amount < `loan_.currentPrincipal + interestRounded_`.

MasterLender.sol

```
288 function _repayPartial(
289     address paymentSource_,
290     Loan memory loan_,
291     uint256 amount_,
292     uint256 interest_
293 ) private {
294     // Payment Variables
295     uint256 principalPayment_ = amount_;
296     uint256 interestPayment_ = 0;
297
298     // If the amount_ is larger than principal then it covers the full
currentPrincipal
299     // and it covers partial payment of interest
300     if (amount_ > loan_.currentPrincipal) {
301         principalPayment_ = loan_.currentPrincipal;
302         interestPayment_ = amount_ - loan_.currentPrincipal;
303     }
304     // Store outstanding interest, extract any paidup interest
305     dataVault.loanRegistry().increaseInterestLocker(
```

```
        loan_.wallet,
        (interest_ - interestPayment_)
    );

    // Make Payment
    _makePayment(paymentSource_, principalPayment_, interestPayment_, 0);

    // Calibrate other contract according to repayment amount
    _calibrateRepayment(loan_, principalPayment_);

    // Emit Event
    emit Repaid(loan_.wallet, paymentSource_, amount_);
}
```

5.16.2. Remediation

Inspex suggests modifying the `_repayPartial()` function as shown in lines 305 - 311, for example:

MasterLender.sol

```
288 function _repayPartial(
289     address paymentSource_,
290     Loan memory loan_,
291     uint256 amount_,
292     uint256 interest_
293 ) private {
294     // Payment Variables
295     uint256 principalPayment_ = amount_;
296     uint256 interestPayment_ = 0;
297 
298     // If the amount_ is larger than principal then it covers the full
currentPrincipal
299     // and it covers partial payment of interest
300     if (amount_ > loan_.currentPrincipal) {
301         principalPayment_ = loan_.currentPrincipal;
302         interestPayment_ = amount_ - loan_.currentPrincipal;
303     }
304 
305     if(interest_ > interestPayment_){
306         // Store outstanding interest, extract any paidup interest
307         dataVault.loanRegistry().increaseInterestLocker(
308             loan_.wallet,
309             (interest_ - interestPayment_)
310         );
311     }
312 
313     // Make Payment
314     _makePayment(paymentSource_, principalPayment_, interestPayment_, 0);
```

```

    // Calibrate other contract according to repayment amount
    _calibrateRepayment(loan_, principalPayment_);

    // Emit Event
    emit Repaid(loan_.wallet, paymentSource_, amount_);
}
```

5.17. Insufficient Logging for Privileged Functions

ID	IDX-017
Target	DataVault DeveloperTreasury Governance InterestRateCalculator MasterLiquidator RevenueDistribution RewardLocker Treasury
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The MoneySwitch team has resolved this issue as suggested by emitting events for the execution of privileged functions in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.17.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the base interest rate by executing the `setBaseRate()` function in the `InterestRateCalculator` contract, and no events are emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function
CreditProtectionPool.sol (L:47)	CreditProtectionPool	moveCollateral()
DataVault.sol (L:57)	DataVault	setGovernance()
DataVault.sol (L:77)	DataVault	setBorrowerRegistry()

DataVault.sol (L:95)	DataVault	setLoanRegistry()
DataVault.sol (L:118)	DataVault	setRevenueDistribution()
DataVault.sol (L:136)	DataVault	setTreasury()
DataVault.sol (L:156)	DataVault	setDeveloperTreasury()
DataVault.sol (L:179)	DataVault	setCreditProtectionPool()
DataVault.sol (L:196)	DataVault	setPoolRegistry()
DataVault.sol (L:214)	DataVault	setMasterLiquidator()
DataVault.sol (L:232)	DataVault	setMasterLender()
DataVault.sol (L:250)	DataVault	setMasterRewards()
DataVault.sol (L:272)	DataVault	setDistributionTreasury()
DataVault.sol (L:293)	DataVault	setInterestRateCalculator()
DeveloperTreasury.sol (L:28)	DeveloperTreasury	requestTransfer()
Governance.sol (L:113)	Governance	setPendingAdmin()
Governance.sol (L:122)	Governance	acceptAdmin()
Governance.sol (L:132)	Governance	setPendingGovernor()
Governance.sol (L:141)	Governance	acceptGovernor()
Governance.sol (L:176)	Governance	addContract()
Governance.sol (L:185)	Governance	deleteContract()
InterestRateCalculator.sol (L:39)	InterestRateCalculator	setBaseRate()
InterestRateCalculator.sol (L:47)	InterestRateCalculator	setCapRate()
InterestRateCalculator.sol (L:55)	InterestRateCalculator	setCapDuration()
InterestRateCalculator.sol (L:73)	InterestRateCalculator	setMaxLoanDuration()
MasterLiquidator.sol (L:29)	MasterLiquidator	suspendLoan()
MasterLiquidator.sol (L:54)	MasterLiquidator	liquidateLoan()
RevenueDistribution.sol (L:85)	RevenueDistribution	updateDistribution()
RewardLocker.sol (L:34)	RewardLocker	setRewardScale()

Treasury.sol (L:22)	Treasury	transferFromTreasury()

5.17.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

InterestRateCalculator.sol

```
event SetBaseRate(uint256 baseRate_);
function setBaseRate(uint256 baseRate_) external onlyAdmin(msg.sender) {
    _baseRate = baseRate_;
    emit SetBaseRate(baseRate_);
}
```

5.18. Unsafe Token Transfer

ID	IDX-018
Target	CreditProtectionPool MasterLender FeederPool MasterPool RevenueDistribution DeveloperTreasury DistributionTreasury Treasury
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The MoneySwitch team has resolved this issue as suggested by replacing the transfer function with functions from `OpenZeppelin's SafeERC20` contract in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.18.1. Description

ERC20 tokens can be improperly implemented, allowing the execution of failed `transfer()` and `transferFrom()` functions without reverting when the invalid transfer amount occurs. However, the tokens in the contracts can only be set by the controllable privileged.

The following table contains all functions that use `transfer()` and `transferFrom()` functions.

Target	Function
CreditProtectionPool.sol (L:31)	reimbursePool()
CreditProtectionPool.sol (L:47)	moveCollateral()
MasterLender.sol (L:327)	_makePayment()
FeederPool.sol (L:63)	deposit()
FeederPool.sol (L:219)	_withdraw()
MasterPool.sol (L:175)	deposit()

MasterPool.sol (L:311)	depositFeeder()
MasterPool.sol (L:335)	withdrawFeeder()
MasterPool.sol (L:380)	borrow()
RevenueDistribution.sol (L:47)	distribute()
DeveloperTreasury.sol (L:28)	requestTransfer()
DistributionTreasury.sol (L:29)	transferTokens()
Treasury.sol (L:22)	transferFromTreasury()

5.18.2. Remediation

Inspex suggests replacing the `transfer()` and `transferFrom()` functions of the tokens with `safeTransfer()` and `safeTransferFrom()` functions from `OpenZeppelin's SafeERC20` contract, for example:

Treasury.sol.sol

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.8.16;

import "../interfaces/IDataVault.sol";
import "../interfaces/ITreasury.sol";
import "../Vaultable.sol";
import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
/// @title Treasury - Holds platform revenue generated meant for MST allocation
contract Treasury is Vaultable, ITreasury {
    using SafeERC20 for IERC20;
    /**
        @dev    Constructor fundction.
    */
    constructor(IDataVault dataVault_) Vaultable(dataVault_) {}

    /**
        @dev    Transfer treasury holdings through two-teir approval process
        @param  liquidityAsset_ Address of external ERC20 contract
        @param  to_ Address of reciving wallet
        @param  amount_ The amount to be transfered
    */
    function transferFromTreasury(
        address liquidityAsset_,
        address to_,
        uint256 amount_
    ) external onlyAllGovernance {
```

```
        //  Make Transfer
        IERC20(liquidityAsset_).safeTransfer(to_, amount_);
    }
}
```

5.19. Inexplicit Solidity Compiler Version

ID	IDX-019
Target	DataVault Vaultable Governance Multiownable BorrowerRegistry CreditProtectionPool LoanRegistry LoanWallet InterestRateCalculator MasterLender MasterLiquidator FeederPool MasterPool PoolRegistry RevenueDistribution MasterRewards RewardLocker DeveloperTreasury DistributionTreasury Treasury
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The MoneySwitch team has resolved this issue as suggested by fixing the Solidity compiler to the latest stable version in commit `48f7a872353c957809239676615e0920d3eb3b95`.

5.19.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

The following table contains all contracts which the inexplicit compiler declares:

File	Version
DataVault.sol (L:2)	^0.8.13
Vaultable.sol (L:2)	^0.8.13
Governance.sol (L:2)	^0.8.13
Multiownable.sol (L:2)	^0.8.13
BorrowerRegistry.sol (L:2)	^0.8.13
CreditProtectionPool.sol (L:2)	^0.8.13
LoanRegistry.sol (L:2)	^0.8.13
LoanWallet.sol (L:2)	^0.8.13
InterestRateCalculator.sol (L:2)	^0.8.13
MasterLender.sol (L:2)	^0.8.13
MasterLiquidator.sol (L:2)	^0.8.13
FeederPool.sol (L:2)	^0.8.13
MasterPool.sol (L:2)	^0.8.13
PoolRegistry.sol (L:2)	^0.8.13
RevenueDistribution.sol (L:2)	^0.8.13
MasterRewards.sol (L:2)	^0.8.13
RewardLocker.sol (L:2)	^0.8.13
DeveloperTreasury.sol (L:2)	^0.8.13
DistributionTreasury.sol (L:2)	^0.8.13
Treasury.sol (L:2)	^0.8.13

5.19.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.17 (https://github.com/ethereum/solidity/releases).

Treasury.sol

```
2 pragma solidity 0.8.17;
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE