BAIL security

SMARDEX
P2P Lending

FINAL REPORT

February ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	SMARDEX P2P Lending
Website	smardex.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SmarDex-Ecosystem/SPRO_contracts/commit/a8946b44b5cb70f7a8c7a18356a3bc7b91299de4
Resolution 1	https://github.com/SmarDex-Ecosystem/SPRO_contracts/tree/c999c795db2e8e5a583e1b4e0368864689ecfb85/src/spro

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	2	1	1		
Medium	1	1			
Low	3	2	1		
Informational	11	8		3	
Governance					
Total	17	12	2	3	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

SproStorage

P2P Lending

The Spro contract is the core contract for the decentralized fixed-rate peer-to-peer (P2P) lending protocol. It facilitates loans by allowing borrowers to create collateralized loan proposals and lenders to fund these proposals.

The contract manages the entire loan lifecycle, including creation, repayment, and liquidation. Borrowers deposit collateral to create proposals, which lenders can then accept by loaning credit tokens. Each loan is represented by an NFT issued to the lender, which can be transferred to other users. Upon repayment of the principal plus fixed interest, the borrower regains their collateral. If a loan defaults due to non-repayment within the stipulated time, the lender can claim the collateral.

Core Invariants

INV 1: A proposal's credit used must never exceed its available credit limit

INV 2: Loan collateral and fixed interest amount must be proportional to the credit amount

INV 3: Each loan must have exactly one SproLoan NFT minted to represent ownership

INV 4: Only the loan token holder can claim a loan's assets after repayment or default

INV 5: A loan can only be repaid if it's in RUNNING state and not expired

Privileged Functions

- transferOwnership
- acceptOwnership
- setFee
- setPartialPositionPercentage
- setLoanMetadataUri

Issue_01	Creating a loan via PERMIT2 results in funds being stuck in the contract
Severity	High
Description	The createLoan function uses the _permit2Workflows function to handle the transfer of the loan credit amount, creditAmount. However, instead of transferring the funds to the borrower, the function incorrectly transfers them to the Spro contract. *PERMIT2.permit(msg.sender, permitSign, data);* *PERMIT2.transferFrom(msg.sender, address(this), amount, token);* This results in the funds being stuck within the contract, leaving the borrower without the intended loan amount. Consequently, the borrower must repay the loan to retrieve their collateral, leading to a potential loss for the lender as the borrower received no loan.
Recommendations	Consider modifying the _permit2Workflows function to ensure that the creditAmount is transferred directly to the borrower, rather than the Spro contract.
Comments / Resolution	Resolved.

Issue_02	Handling of non-standard ERC-20 tokens can lead to accounting issues
Severity	**High**
Description	The Spro contract encounters issues when interacting with non-standard ERC-20 tokens, such as those with fee-on-transfer (FoT) or rebasing mechanisms. These tokens can cause discrepancies in expected balances and lead to insolvency since there will be a discrepancy between the stored balances and the actual token amounts. 1. **Fee-on-transfer tokens**: The contract may not receive the full amount of tokens due to transfer fees, leading to incorrect accounting. 2. **Rebasing tokens**: Escrowed collateral balances (e.g., stETH) may fluctuate unexpectedly, causing reverts when transferring cached balances or transferring too few tokens. 3. **Transfer of less than the specified amount**: Tokens like cUSDCv3 may transfer less than the specified amount, potentially allowing for collateral theft if proposal.collateralAmount is set to type(uint256).max.
Recommendations	To mitigate 1 and 3, consider implementing balance checks before and after token transfers to determine the actual amount of tokens received. For 2, rebasing tokens, as there is no generic mitigation that works for all tokens, consider explicitly documenting that they are not supported, and cap the transfer amounts to the contract's actual token balances.
Comments / Resolution	Partially resolved. The client acknowledges that rebasing tokens are not supported. The use of such tokens will lead to accounting issues. 1. and 3. are mitigated by adding balance changes before and after token transfers, strictly enforcing the received token amount to equal the expected amount, and reverting otherwise.

	However, this prevents the use of fee-on-transfer tokens as the received token amount will be less than expected.

Issue_03	PERMIT2 call is vulnerable to front-running and can be used for griefing attacks
Severity	**Medium**
Description	The PERMIT2 permit function call in both the createProposal and _permit2Workflows functions can be front-run by an attacker who intercepts the signature and directly calls the permit function on Uniswap's PERMIT2 contract. As a result, the permit nonce cannot be re-used by the user's legitimate transaction, causing the transaction to revert. This can be exploited by an attacker to grief Spro users who intend to use the permit mechanism instead of using an approval transaction to approve the Spro contract as a token spender.
Recommendations	Consider adding a try-catch block around the permit function call to handle potential reverts gracefully and prevent the transaction from failing.
Comments / Resolution	Resolved.

Issue_04	The cancelProposal function allows the cancellation of non-existent proposals
Severity	Low
Description	The cancelProposal function in the Spro contract does not verify whether a proposal exists before attempting to cancel it. This oversight allows the function to be called with an arbitrary, non-existent proposal, resulting in a no-operation that performs a zero-value collateral token transfer and emits events. This behavior could lead to off-chain monitoring difficulties.
Recommendations	Consider adding a check to the cancelProposal function to ensure the proposal hash exists in storage before proceeding with the cancellation process.
Comments / Resolution	Resolved.

Issue_05	Identical proposals can be re-created after cancellation, which might have incorrect _creditUsed accounting
Severity	Low
Description	When a proposal is canceled with the cancelProposal function, the _proposalsMade storage mapping is set to false, allowing the same proposal with the same hash to be created again. However, the _creditUsed mapping is not cleared, which means the new proposal will have a reduced credit capacity due to the non-zero _creditUsed value. As a result, the collateral cannot be fully utilized, and the proposal must be canceled to refund the unusable collateral to the borrower.
Recommendations	Consider clearing the _creditUsed storage mapping when a proposal is canceled. Additionally, consider preventing repeatedly canceling proposals by keeping track of canceled proposals instead of deleting them from storage.
Comments / Resolution	Partially resolved. The added auto-incrementing nonce will ensure unique proposal hashes. However, although unlikely, a hash collision is still possible. Therefore, it is recommended to explicitly check if the proposal with the calculated hash already exists, by checking _proposalsMade[proposalHash].

Issue_06	During proposal creation, the proposer and partialPositionBps fields are overwritten, which might unexpectedly change proposal parameters
Severity	**Low**

Description	In the createProposal function, the caller-provided Proposal.proposer and Proposal.partialPositionBps fields are overwritten in the code, which can lead to inconsistencies between the hash calculated from the caller-provided proposal and the subsequently stored hash. This might be problematic when an integrating third-party contract uses a pre-calculated proposal hash for further processing.
	proposal.partialPositionBps = _partialPositionBps;
	proposal.proposer = msg.sender;
	Moreover, this discrepancy could result in unexpected behavior, such as a borrower creating a proposal that is front-run by an admin transaction that changes the partialPositionBps value.

Recommendations	Instead of overwriting these values, consider enforcing that the provided values match the expected values, e.g., using require statements. This will ensure consistency and prevent unexpected changes to proposal parameters.
Comments / Resolution	Resolved.

Issue_07	Potential for unusable proposals with past timestamps
Severity	**Informational**
Description	Both the loanExpiration and startTimestamp proposal timestamps can be set to a past timestamp, as long as loanExpiration is greater than startTimestamp. However, this results in a proposal that cannot accept loans due to checking the proposal's expiry and must be canceled to refund the borrower's collateral.
Recommendations	Consider adding a check to ensure that startTimestamp is set to a future timestamp relative to the current block time.
Comments / Resolution	Resolved.

Issue_08	Inconsistent fee parameter validation in constructor
Severity	**Informational**
Description	The constructor of the Spro contract allows the fee parameter to be set without the same validations present in the setFee function. This inconsistency can lead to arbitrary fee values being set during contract deployment, potentially exceeding the MAX_SDEX_FEE maximum bound.
Recommendations	Consider applying the same validation logic for the fee parameter in the constructor used in the setFee function.
Comments / Resolution	Resolved.

Issue_09	Unintuitive credit amount validation logic
Severity	**Informational**
Description	The validation logic for the remaining credit amount in the _acceptProposal function is currently written as *proposal.availableCreditLimit - minAmount < total* which can be unintuitive to read. This logic checks that the leftover credit is at least the minimum amount required for subsequent lending.
Recommendations	Consider rewriting the validation logic as *proposal.availableCreditLimit - total < minAmount* to improve readability and clarity.
Comments / Resolution	Resolved.

Issue_10	Pre-calculation of minAmount when creating a proposal
Severity	**Informational**
Description	The minAmount value, which represents the minimum credit amount, is calculated whenever a proposal is accepted, i.e., a loan created, via the _acceptProposal function. However, all necessary values are known when calling the _makeProposal function during proposal creation, allowing for pre-calculation and gas savings.
Recommendations	Consider calculating minAmount in the _makeProposal function to save gas.
Comments / Resolution	Resolved.

Issue_11	Unused PERMIT2 allowance
Severity	**Informational**
Description	When using the PERMIT2 functionality, users may provide a larger allowance than required, which remains unused and cannot be utilized within the contract, as it is always required to provide a valid permit signature. This can lead to dangling allowances.
Recommendations	Consider adding functionality to utilize existing PERMIT2 allowances, ensuring that provided allowances can be fully utilized.
Comments / Resolution	Acknowledged.

Issue_12	Lack of reentrancy protection in the claimLoan function
Severity	**Informational**
Description	The claimLoan function currently lacks a nonReentrant modifier, which is a safety precaution to prevent reentrancy attacks. Although there is no immediate risk, adding this modifier would be a safety precaution.
Recommendations	Consider adding the nonReentrant modifier to the claimLoan function to prevent potential reentrancy vulnerabilities.
Comments / Resolution	Resolved.

Issue_13	The totalLoanRepaymentAmount function should skip loans with the status NONE instead of returning early with a zero value
Severity	**Informational**
Description	The totalLoanRepaymentAmount function is supposed to return the total credit amount to be repaid for the provided loan IDs. However, the function discards previous calculations and returns 0 when it encounters a loan with a NONE status.
Recommendations	Consider skipping the NONE status loan in this case by using continue, so that the caller can still get an estimate of the total credit. Alternatively, revert if this is not desired so it is clear that there is an issue.
Comments / Resolution	Resolved. The function now only considers repayable loans.

Issue_14	The repayMultipleLoans function will not throw an error when a loanId is not repayable
Severity	**Informational**
Description	The repayMultipleLoans function performs the following check to ensure that a loan is repayable: *// Checks: loan can be repaid & credit address is the same for all loanIds* *if (_isLoanRepayable(loan.status, loan.loanExpiration)) {* However, there is no else statement. Consequently, it will silently ignore loans that were not repaid.
Recommendations	Consider throwing an error when a loan is not repayable.
Comments / Resolution	Acknowledged.

Issue_15	A loan cannot be repaid if the borrower gets blacklisted from spending the collateral token
Severity	**Informational**
Description	In the repayLoan, when the collateral token is USDC, if the borrower gets blacklisted, the loan will not be repayable due to reverting when attempting to transfer: *IERC20Metadata(loan.collateral).safeTransfer(loan.borrower, loan.collateralAmount);*
Recommendations	If the caller of the repayLoan function is the borrower, consider adding the option for the caller to specify the collateral recipient address. Alternatively, the issue could be acknowledged.
Comments / Resolution	Resolved.

SproLoan

The SproLoan contract manages loans as NFTs. It allows the minting and burning of loan tokens, which represent individual loans, and the owner to manage the base metadata URI for these tokens.

The contract is deployed during the Spro contract deployment, and the owner is set to the Spro contract address to ensure that only this contract can mint and burn loan tokens.

Privileged Functions

- transferOwnership
- renounceOwnership
- mint
- burn
- setLoanMetadataUri

Issue_16	Override the _baseURI function in the SproLoan contract
Severity	**Informational**
Description	In the SproLoan contract, the inherited tokenURI function is overridden and returns based on the _metadataUri storage variable the URI for the given token. However, this can be simplified by overriding the _baseURI function, eliminating the need for a custom tokenURI implementation, and following the best practice.
Recommendations	Consider overriding the _baseURI function to provide the base URI for tokens.
Comments / Resolution	Acknowledged.

Issue_17	The _safeMint function might prevent some EOAs from creating a loan
Severity	**Informational**
Description	The current _safeMint function initiates an onERC721Received callback if the to address is a contract (code.length >0) . However, the issue is that with the Pectra Ethereum upgrade, the code.length of EOAs can be greater than 0 as well, causing the sproLoan NFT mint on createLoan to fail in such scenarios if the EOA's code is missing this required callback function.
Recommendations	Best to acknowledge the issue as _safeMint has its advantages. Alternatively, using the regular _mint function can be used to eliminate callbacks and thus this problem.
Comments / Resolution	Resolved. _mint is now used instead of _safeMint.