Kapan Finance - Lending Aggregator

SECURITY ASSESSMENT REPORT

September 2, 2025

Prepared for Kapan Finance

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the results of a security assessment conducted by CODESPECT for Kapan Finance. Kapan is a decentralized lending aggregator enabling seamless interaction with multiple lending protocols through a single interface.

The scope of this audit includes the Cairo-based Kapan Finance contracts, specifically the router responsible for interacting with protocol-specific gateways. These gateways interact with their corresponding lending protocols.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of programs, execution testing.

CODESPECT found ten points of attention, two classified as `High`, three classified as `Medium`, three classified as `Low`, and two classified as `Best Practices`. All of the issues are summarised in Table 2.

Audit Conclusion

CODESPECT conducted a thorough security review of the Kapan Finance smart contracts. All severe issues identified during the audit were addressed by the Kapan team. The fix review process, however, required several rounds and included longer intervals between submissions. In addition, some fixes introduced notable logical changes to the code.
Given these circumstances, we strongly recommend a secondary review to ensure the overall robustness and security of the contracts.

Organisation of the document is as follows:

- **Section 5** summarises the audit.
- **Section 6** describes the system overview.
- **Section 7** presents the issues.
- **Section 8** contains additional notes for the audit.
- **Section 9** discusses the documentation provided by the client for this audit.
- **Section 10** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
High	0	2	0
Medium	0	3	0
Low	0	2	1
Best Practices	0	2	0
Total	**0**	**9**	**1**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	Kapan Finance - Lending Aggregator
Type of Project	Lending Aggregator
Duration of Engagement	6 Days
Duration of Fix Review Phase	3 Days
Draft Report	June 26, 2025
Final Report	September 2, 2025
Repository	kapan
Commit (Audit)	83a7747df3350c2b23d747d52bdd998adbc8812d
Commit (Final)	a717a11c18aa8fb9bbb3732b96dc04ad091ba69a
Documentation Assessment	Medium
Test Suite Assessment	Medium
Auditors	Kalogerone, shaflow01

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	LoC
1	kapan/packages/snfoundry/contracts/src/lib.cairo	18
2	kapan/packages/snfoundry/contracts/src/gateways/NostraGateway.cairo	303
3	kapan/packages/snfoundry/contracts/src/gateways/vesu_gateway.cairo	708
4	kapan/packages/snfoundry/contracts/src/gateways/RouterGateway.cairo	343
	Total	**1372**

5.2 Findings Overview

	Finding	Severity	Update
1	Certain combinations of instructions can lead to token loss	High	Fixed
2	Vesu Gateway uses same the default pool id for every withdrawal	High	Fixed
3	Certain instruction combos create negative balancesAfter and will revert	Medium	Fixed
4	Repay may fail due to insufficient tokens approval	Medium	Fixed
5	The on_flash_loan function lacks a caller verification check	Medium	Fixed
6	Nostra positions are not getting tracked correctly	Low	Acknowledged
7	The on_flash_loan function does not take the repay_all flag into account when handling the repay instruction	Low	Fixed
8	get_flash_loan_amount may fail to return the correct amount	Low	Fixed
9	Revoke excess approvals in after_send_instructions	Best Practices	Fixed
10	Some transfers don't confirm the return boolean	Best Practices	Fixed

6 System Overview

The Kapan Finance protocol is a DeFi routing and aggregation protocol built on Starknet that enables seamless interaction with multiple lending protocols through a unified interface. The protocol acts as an intermediary layer that abstracts protocol-specific complexities, allowing users to execute complex cross-protocol strategies and optimise yields across different lending platforms in atomic transactions.

The protocol follows a hub-and-spoke architecture centred around the `RouterGateway` contract, which serves as the primary orchestrator for all lending operations. This central hub communicates with protocol-specific gateways that act as adapters for underlying lending protocols and never holds any funds.

Currently, the system integrates with two major lending protocols through dedicated gateways: The `Vesu Gateway` interacts with the Vesu lending protocol, supporting isolated lending pairs with variable interest rates, while `Nostra Gateway` connects to the Nostra protocol, which offers pooled lending with interest-bearing receipt tokens. Users can interact with the system through the `RouterGateway` contract for cross-protocol operations and complex strategies, or directly with individual protocol gateways when operating within a single lending protocol.

The system defines four primary instructions that mirror standard lending operations. Each basic instruction contains three essential components: the token address, the amount, and the user address:

- **Deposit** instructions add collateral to a lending position.
- **Withdraw** instructions remove collateral from lending positions.
- **Borrow** instructions create debt against deposited collateral.
- **Repay** instructions reduce or eliminate outstanding debt.

Beyond basic operations, Kapan Finance introduces two innovative instruction types that enable sophisticated chaining of operations:

- **Reborrow** instructions dynamically borrow an amount determined by a previous operation in the same transaction.
- **Redeposit** instructions work similarly, depositing an amount determined by a previous operation.

Administrative functions in the Kapan Finance protocol are carefully scoped to configuration and expansion capabilities without compromising user fund security. The `RouterGateway` owner possesses the critical ability to register new protocol gateways, enabling the system to expand to additional lending protocols.

Each protocol gateway maintains its own administrative functions tailored to the specific requirements of the underlying protocol. The Vesu Gateway administrator can add new supported assets, register additional lending pools, and configure asset-pool relationships. Similarly, the Nostra Gateway administrator manages the mapping between underlying assets and their corresponding debt and collateral token representations within the Nostra protocol.

7 Issues

7.1 [High] Certain combinations of instructions can lead to token loss

File(s): RouterGateway.cairo

Description: Through the RouterGateway contract, users can choose to execute multiple sequential instructions on a single gateway. Before executing the instructions, the contract checks its token balances. After execution, it calculates the balance changes and transfers tokens to the user accordingly. However, since the pre-execution balance includes tokens that are meant to be input (e.g., for deposit or repay), certain combinations of instructions may result in token loss.

```
fn before_send_instructions(...) -> Span<u256> {
    let mut i: usize = 0;
    let mut balancesBefore = array![];
    while i != instructions.len() {
        match instructions.at(i) {
            LendingInstruction::Deposit(deposit) => {
                let basic = *deposit.basic;
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                if should_transfer {
                    assert(erc20.transfer_from(get_caller_address(), get_contract_address(), basic.amount), 'transfer
                    ↪ failed');
                }
                assert(erc20.approve(gateway, basic.amount), 'approve failed');
                let balance = erc20.balance_of(get_contract_address());
                balancesBefore.append(balance);
            },
            LendingInstruction::Repay(repay) => {
                let basic = *repay.basic;
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                if should_transfer {
                    assert(erc20.transfer_from(get_caller_address(), get_contract_address(), basic.amount), 'transfer
                    ↪ failed');
                }
                assert(erc20.approve(gateway, basic.amount), 'approve failed');
                let balance = erc20.balance_of(get_contract_address());
                balancesBefore.append(balance);
            },
            //...
```

Some combinations of instructions may lead to token loss. For example:

1. repay token1 with 100;
2. withdraw to retrieve 110 of token1;

Since 100 token1 tokens are input into the contract in advance, balanceBefore = [100, 100] During execution, 100 token1 tokens are used to repay the debt, and the withdraw retrieves 110 tokens. balanceAfter = 110 - 100 = 10 Only 10 token1 tokens are sent to the user, while the remaining 100 tokens remain locked in the contract.

Impact: Certain instructions combinations that use the same token can lead to permanent loss.

Recommendation(s): It is recommended that after_send_instructions fetch the token balances before any token inputs occur, and then iterate through the instructions to execute token inputs.

Status: Fixed

Update from Kapan:

7.2 [High] Vesu Gateway uses the same default pool ID for every withdrawal

File(s): `vesu_gateway.cairo`

Description: During withdrawals from Vesu, users specify from which pool they want to withdraw using the `context` field in the `Withdraw` struct:

```
fn withdraw(ref self: ContractState, instruction: @Withdraw) {
    // ...
    if instruction.context.is_some() {
        let mut context_bytes: Span<felt252> = (*instruction.context).unwrap();
        let vesu_context: VesuContext = Serde::deserialize(ref context_bytes).unwrap();
        if vesu_context.pool_id != Zero::zero() {
            pool_id = vesu_context.pool_id;
        }
        if vesu_context.position_counterpart_token != Zero::zero() {
            debt_asset = vesu_context.position_counterpart_token;
        }
    }
    // ...
```

Later, contract needs to call the correct `vToken` address to convert user's shares to assets. However, the `vToken` retrieved is not from the user's specified `pool_id`:

```
fn modify_collateral_for(
    ref self: ContractState,
    pool_id: felt252,
    collateral_asset: ContractAddress,
    debt_asset: ContractAddress,
    user: ContractAddress,
    collateral_amount: i257,
) -> UpdatePositionResponse {
    // ...
    // If this is negative, it means withdraw
    if collateral_amount.is_negative() {
        // @audit doesn't pass user's pool id
        let vtoken = self.get_vtoken_for_collateral(collateral_asset);

        let erc4626 = IERC4626Dispatcher { contract_address: vtoken };
        let requested_shares = erc4626.convert_to_shares(collateral_amount.abs());
        let available_shares = vesu_context.position.collateral_shares;
        assert(available_shares > 0, 'No-collateral');
        // ...
}
```

```
fn get_vtoken_for_collateral(
    self: @ContractState, collateral: ContractAddress,
) -> ContractAddress {
    let vesu_singleton_dispatcher = ISingletonDispatcher {
        contract_address: self.vesu_singleton.read(),
    };
    // @audit uses contract's default pool id
    let poolId = self.pool_id.read();
    let extensionForPool = vesu_singleton_dispatcher.extension(poolId);
    let extension = IDefaultExtensionCLDispatcher { contract_address: extensionForPool };
    extension.v_token_for_collateral_asset(poolId, collateral)
}
```

As a result, the wrong `vToken` address is used to retrieve information about the user's available shares and final withdraw amount.

Impact: Users are unable to withdraw from their required pools as the transaction will revert if they don't have any shares in the default `pool_id`. Also, users who have shares in that pool will withdraw assets from that pool even if they specified another pool.

Recommendation(s): Pass users' `pool_id` to the `get_vtoken_for_collateral(...)` function and use that to retrieve the extension contract address.

Status: Fixed

Update from Kapan:

7.3 [Medium] Certain instruction combos create negative `balancesAfter` and will revert

File(s): `RouterGateway.cairo`

Description: Through the `RouterGateway` contract, users can choose to execute multiple sequential instructions on a single gateway. Before executing the instructions, the contract checks its token balances. After execution, it checks its token balances again and calculates the difference from the first check. However, this difference may be a negative value, which will result in the transaction reverting since `balancesAfter` is an array of `u256`.

For example:

1. Deposit `100` of `token1`;
2. Borrow `100` of `token1`;

Since `100` tokens are transferred into the contract in advance, `balanceBefore = [100, 100]`

Looking at `balancesAfter` calculation for these 2 instructions:

```
fn after_send_instructions(
    ref self: ContractState,
    gateway: ContractAddress,
    instructions: Span<LendingInstruction>,
    balancesBefore: Span<u256>,
    should_transfer: bool,
) -> Span<u256> {
    let mut i: usize = 0;
    let mut balancesAfter = array![];
    while i != instructions.len() {
        match instructions.at(i) {
            LendingInstruction::Borrow(borrow) => {
                let basic = *borrow.basic;
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                if should_transfer {
                    assert(
                        erc20
                            .transfer(basic.user, erc20.balance_of(get_contract_address())),
                        'transfer failed',
                    );
                }
                let balance = erc20.balance_of(get_contract_address());

                balancesAfter.append(balance - *balancesBefore.at(i));
            },
            ...

            LendingInstruction::Deposit(deposit) => {
                let basic = *deposit.basic;
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                let balance = erc20.balance_of(get_contract_address());
                balancesAfter.append(*balancesBefore.at(i) - balance);
            },
            _ => {},
        }
        i += 1;
    }
    balancesAfter.span()
}
```

Here `Borrow` will first transfer the `100` borrowed tokens to the user and then track the balance of the contract. As a result, `balanceAfter` here will be attempted to be `0 - 100` and transaction will revert.

Impact: Certain instruction combinations will have a negative value for `balanceAfter` and will result in the transaction reverting.

Recommendation(s):

Status: Fixed

Update from Kapan:

7.4 [Medium] Repay may fail due to insufficient tokens approval

File(s): `NostraGateway.cairo`, RouterGateway.cairo

Description: If the `repay_all` field is enabled in the repay instruction, it is expected to repay all outstanding debt. However, since this value does not overwrite the `amount` field in the `repay` instruction within `instructions`.

```
fn before_send_instructions(...) -> Span<u256> {
    //...
            LendingInstruction::Repay(repay) => {
                let basic = *repay.basic;
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                if should_transfer {
                    assert(erc20.transfer_from(get_caller_address(), get_contract_address(), basic.amount), 'transfer
                     → failed');
                }
                assert(erc20.approve(gateway, basic.amount), 'approve failed');
                let balance = erc20.balance_of(get_contract_address());
                balancesBefore.append(balance);
            },
```

Impact: It may result in a failed repayment in `before_send_instructions` due to insufficient approval or token transfer. For example, if the `repay_all` flag is enabled but the `repay.amount` is arbitrarily set to a value like 1, then `before_send_instructions` will not transfer a sufficient amount of tokens to the Router, and the Router will not approve enough allowance to the Gateway, resulting in the repay operation failing.

Recommendation(s): In `before_send_instructions`, before transferring and approving tokens for a repay instruction, check if `repay_all` is enabled. If it is, replace the operation amount with the corresponding total debt amount instead of using `repay.amount`.

Status: Fixed

Update from Kapan:

7.5 [Medium] The `on_flash_loan(...)` function lacks a caller verification check

File(s): RouterGateway.cairo

Description: The `on_flash_loan` function is the flash loan callback function. After receiving the flash loan, the contract executes instruction logic within `on_flash_loan`. However, since there is no check to ensure that the caller is the `flashloan_provider`, a malicious actor can arbitrarily call this function to bypass the `ensure_user_matches_caller` check and execute instructions.

```
fn on_flash_loan(...) {
    assert(sender == get_contract_address(), 'sender mismatch');
    println!("Received flash loan");
    //...
}
```

Impact: This could potentially lead to the theft of tokens that users have approved for the Router contract — For example, if a user wants to withdraw from `NostraGateway` via the router, they need to approve `nibcollateral` to the router. A malicious actor can check for such approvals. Then call `on_flash_loan` to execute the `withdraw` instruction, and then call the `deposit` instruction to steal those tokens.

Recommendation(s): It is recommended to check whether the caller is the flashloan provider.

Status: Fixed

Update from Kapan:

7.6 [Low] Nostra positions are not getting tracked correctly

File(s): `NostraGateway.cairo`

Description: In the Nostra protocol there are 2 types of collateral that users can have, Interest Bearing collateral and Non-Interest Bearing collateral. It is possible that users hold both of these debt tokens simultaneously. However, the `get_user_positions(...)` function doesn't account for this scenario:

```
fn get_user_positions(
    self: @ContractState, user: ContractAddress,
) -> Array<(ContractAddress, felt252, u256, u256)> {
    let mut positions = array![];
    let mut i = 0;
    while i != self.supported_assets.len() {
        let underlying = self.supported_assets.at(i).read();
        let symbol = IERC20SymbolDispatcher { contract_address: underlying }.symbol();

        let debt = self.underlying_to_ndebt.read(underlying);
        let collateral = self.underlying_to_ncollateral.read(underlying);
        let ibcollateral = self.underlying_to_nibcollateral.read(underlying);

        let debt_balance = IERC20Dispatcher { contract_address: debt }.balance_of(user);
        let collateral_raw = IERC20Dispatcher { contract_address: collateral }
            .balance_of(user);

        // @audit User can have collateral in both the collateral and ibcollateral tokens
        let collateral_balance = if collateral_raw == 0 {
            IERC20Dispatcher { contract_address: ibcollateral }.balance_of(user)
        } else {
            collateral_raw
        };
        positions.append((underlying, symbol, debt_balance, collateral_balance));
        i += 1;
    };
    return positions;
}
```

The function checks the user's Non-Interest Bearing collateral balance and only if it's `0` it checks for user's Interest Bearing collateral balance.

Impact: The function always returns the balance of 1 type of collateral that the user holds and never the total of both of them. If a user is holding Non-Interest Bearing collateral tokens, his Interest Bearing collateral balance will not be accounted for.

Recommendation(s): Check for both of the balances and add them.

Status: Acknowledged

Update from Kapan: This one won't be fixed in the release version as the UI does not support the nostra semantics. Users would be required to go through their portal to setup the tokens correctly for transfering debt for the time being.

7.7 [Low] The `on_flash_loan(...)` function does not take the `repay_all` flag into account when handling the `repay` instruction

File(s): `RouterGateway.cairo`

Description: In the `on_flash_loan` function, funds are obtained via flash loan to prepare for debt repayment. The total repay amount is recalculated and the repay instruction is reconstructed. However, since the `repay_all` flag is not taken into account, some of these checks may become invalid.

```
fn on_flash_loan(...) {
    //...
    for protocolInstruction in protocol_instructions {
        for instruction in protocolInstruction.instructions {
            if let LendingInstruction::Repay(repay) = instruction {
                let repay = *repay;
                repay_amounts.append(repay.basic.amount);
                total_repay_amount += repay.basic.amount;
                repay_count += 1;
            }
        }
    }

    // Calculate remaining amount to distribute
    let remaining_amount = amount - total_repay_amount;
    assert(remaining_amount >= 0, 'flashloan insufficient');
    //...
    for instruction in protocol_instructions {
            //...
                        basic: BasicInstruction {
                            token: repay.basic.token,
                            amount: modified_amount,
                            user: repay.basic.user,
                        },
                        repay_all: false, // Force explicit amount
                        context: repay.context,

        //...
```

When determining the flash loan amount, if the `repay_all` flag is enabled in the repay instruction, the flash loan amount is treated as the full debt rather than `repay.amount`.

However, in the `on_flash_loan` function, during validation and instruction reconstruction, `repay.amount` is used without handling the `repay_all` flag explicitly. This may lead to invalid or ineffective checks.

Impact: If `repay_all` is enabled and `repay.amount` does not match the full debt amount, then the `move_debt` function may fail.

Recommendation(s): The `on_flash_loan` function should consider the `repay_all` flag when accumulating amounts and reconstructing repay instructions.

Status: Fixed

Update from Kapan:

7.8 [Low] `get_flash_loan_amount(...)` may fail to return the correct amount

File(s): `RouterGateway.cairo`

Description: In the `get_flash_loan_amount(...)` function, if a repay instruction has the `repay_all` flag enabled, the function will immediately return the flash loan amount for the gateway. However, the function does not consider whether the other `ProtocolInstructions` also contains a repay instruction.

```
fn get_flash_loan_amount(...) {
   let mut flash_loan_amount : u256 = 0;
   let mut token : ContractAddress = Zero::zero();
   for protocolInstruction in instructions {
        for instruction in protocolInstruction.instructions {
            if let LendingInstruction::Repay(repay) = instruction {
                assert(*repay.basic.amount != 0, 'repay-amount-is-zero');
                if *repay.repay_all {
                    let gateway = ILendingInstructionProcessorDispatcher { contract_address:
                     ↪  self.gateways.read(*protocolInstruction.protocol_name) };
                    return (*repay.basic.token, gateway.get_flash_loan_amount(*repay));
                }
                flash_loan_amount += *repay.basic.amount;
                token = *repay.basic.token;
            }
        };
   };
   //...
}
```

Impact: If there are multiple `ProtocolInstructions` in the array and one of the repay instructions has the `repay_all` flag enabled, due to the absence of the required `amount` in other repay instructions, the flash loan amount will be insufficient, causing the `move_debt` instruction to fail.

Recommendation(s): When the `repay_all` flag is enabled, do not consider the flash loan amount required for just a single market.

Status: Fixed

Update from Kapan:

7.9 [Best Practice] Revoke excess approvals in `after_send_instructions(...)`

File(s): `RouterGateway.cairo`

Description: In `after_send_instructions`, there may be cases where not all tokens are used during repayment because `repay.amount > debt`. The unused tokens will be transferred from the router to the user. However, the approval for these tokens granted to the gateway in `before_send_instructions` has not yet been revoked.

```
fn after_send_instructions(ref self: ContractState, gateway: ContractAddress, instructions: Span<LendingInstruction>,
↪  balancesBefore: Span<u256>, should_transfer: bool) -> Span<u256> {
// ...
        LendingInstruction::Repay(repay) => {
            let basic = *repay.basic;
            let erc20 = IERC20Dispatcher { contract_address: basic.token };
            let balance = erc20.balance_of(get_contract_address());
            let diff = *balancesBefore.at(i) - balance;
            balancesAfter.append(diff);
            if basic.amount > diff {
                let erc20 = IERC20Dispatcher { contract_address: basic.token };
                erc20.transfer(basic.user, basic.amount - diff);
            }
        },
// ...
}
```

Impact: A user can manipulate the system to cause the router to grant an excessively large approval to the gateway. For example, if `user1` only has a debt of 10 but sets `repay.amount` to 10,000 during repayment, the unused 9,990 tokens will be returned to `user1`. However, the router's approval of 9,990 tokens to the gateway remains in place.

While this may not have an immediate visible impact, it is recommended to revoke this unused approval to reduce the potential attack surface in future updates.

Recommendation(s): It is recommended to revoke the router's approval to the gateway for the refunded tokens.

Status: Fixed

Update from Kapan:

7.10 [Best Practice] Some transfers don't confirm the return boolean

Description: Throughout the protocol, `transfer(...)` and `transfer_from(...)` functions returns are checked to be `true`. However, there are a few instances where this doesn't happen:

a. `RouterGateway.cairo`:
   - In `after_send_instructions(...)` function under `Withdraw` and `Repay` instructions.

b. `NostraGateway.cairo`:
   - In `repay(...)` function when transferring the `underlying_token`.

c. `vesu_gateway.cairo`:
   - In `repay(...)` function when transferring any `remainder` amount.

Status: Fixed

Update from Kapan:

8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- Supported assets are not enforced to the users instructions in `vesu_gateway.cairo` but they are used in the view/UI functions which will potentially miss some user positions.
- The `vesu_gateway.cairo` contract should use the `ISingletonV2` Interface as this is the one that it's interacting with.
- In the `vesu_gateway.cairo` contract there is an inconsistency that `repay(...)` and `borrow(...)` functions require that the instructions `context` is not empty, but the `deposit(...)` and `withdraw(...)` functions don't.

9 Evaluation of Provided Documentation

The **Kapan Finance** documentation was provided in a single form:

- **Natspec Comments:** Some parts of the code included Natspec comments, which explained the purpose of complex functionality in detail and facilitated understanding of individual functions. However, some functionalities lacked comments, and expanding documentation coverage would enhance the overall comprehensibility of the code.

The documentation provided by **Kapan Finance** offered valuable insights into the protocol, significantly aiding CODESPECT's understanding. However, the public technical documentation could be further improved to better present the protocol's overall functionality and facilitate the understanding of each component.

Additionally, the **Kapan Finance** team provided a technical walkthrough of the codebase and was consistently available and responsive, promptly addressing all questions raised by CODESPECT during the evaluation process.

10 Test Suite Evaluation

10.1 Compilation Output

```
% scarb build
Compiling snforge_scarb_plugin v0.45.0
Finished `release` profile [optimized] target(s) in 0.07s
Compiling kapan v0.0.1

warn: Unused import: `kapan::gateways::vesu_gateway::ByteArrayTrait`
 --> \textbf{Kapan Finance}-Starknet/packages/snfoundry/contracts/src/gateways/vesu_gateway.cairo:2:23
use core::byte_array::ByteArrayTrait;
                      ^************^

// The rest of the compilation output with warnings was omitted

Finished `dev` profile target(s) in 38 seconds
```

10.2 Tests Output

```
% snforge test
Compiling snforge_scarb_plugin v0.45.0
Finished `release` profile [optimized] target(s) in 0.03s
Compiling test(kapan_unittest) kapan v0.0.1

Finished `dev` profile target(s) in 38 seconds
Collected 18 test(s) from kapan package
Running 0 test(s) from src/
Running 18 test(s) from tests/
[IGNORE] kapan_integrationtest::TestNostra::test_deploy_and_add_supported_assets
[IGNORE] kapan_integrationtest::TestNostra::test_deposit
[IGNORE] kapan_integrationtest::TestNostra::test_full_flow
[IGNORE] kapan_integrationtest::TestNostra::test_get_borrow_rate
[IGNORE] kapan_integrationtest::TestNostra::test_get_interest_rates
[IGNORE] kapan_integrationtest::TestNostra::test_get_user_positions
[IGNORE] kapan_integrationtest::TestNostra::test_withdraw
[IGNORE] kapan_integrationtest::TestRouter::test_move_debt_reverse
[IGNORE] kapan_integrationtest::TestRouter::test_router_setup
[IGNORE] kapan_integrationtest::TestRouter::test_vesu
[IGNORE] kapan_integrationtest::TestVesu::test_basic_withdraw
[IGNORE] kapan_integrationtest::TestVesu::test_borrow
[IGNORE] kapan_integrationtest::TestVesu::test_deposit
[IGNORE] kapan_integrationtest::TestVesu::test_get_all_positions
[IGNORE] kapan_integrationtest::TestVesu::test_get_borrow_rate
[IGNORE] kapan_integrationtest::TestVesu::test_get_supported_assets_ui
[IGNORE] kapan_integrationtest::TestVesu::test_repay
Deploying RouterGateway
Deploying NostraGateway
Deploying VesuGateway
Adding supported assets to NostraGateway
Pre-funding user address
Prefunded with token
Result: Result::Ok([1])
Result: Result::Ok([1])
Processing protocol instruction 0
Processed protocol instruction 0
Processing move debt
Requesting flash loan
Received flash loan
Processing protocol instruction 0
Processed protocol instruction 0
Processing protocol instruction 1
Processed protocol instruction 1
[PASS] kapan_integrationtest::TestRouter::test_move_debt (l1_gas: ~0, l1_data_gas: ~5536, l2_gas: ~1212493440)
Tests: 1 passed, 0 failed, 17 ignored, 0 filtered out

Latest block number = 1520484 for url = https://starknet-mainnet.public.blastapi.io/rpc/v0_8
```

The tests are ignored as they need to be executed separately due to endpoint restrictions.

10.3 Notes about Test suite

The **Kapan Finance** team provided a comprehensive test suite consisting of various unit tests that cover the majority of flows and core functionalities. These tests help verify that individual components behave as expected in isolation.

CODESPECT identified an opportunity to enhance test coverage by introducing additional fuzz tests. These tests are designed to validate functionality under unexpected or edge-case inputs, helping to ensure that critical assumptions identified during the manual audit remain valid—thereby strengthening the protocol's overall security and robustness.

Furthermore, **CODESPECT** recommends explicitly defining strict invariants that the protocol must uphold. Implementing targeted tests to validate these invariants would provide continuous assurance that the system behaves correctly across all conditions, further reinforcing both stability and security.