



SMART CONTRACT SECURITY AUDIT

Axios

Lending Markets

CLIENT

Axios Finance

PLATFORM

Fuel

LANGUAGE

Sway

REVIEW

December 2025

DOCUMENT

Document information

Project	Axios, Lending Markets
Client	Axios Finance
Platform	Fuel
Language	Sway
Repository	AxiosFuel/axios-core-contracts
Commit	85c1cb6dfe75ff7d5159cc4916ce4a6bba66e67e
Methods	Manual code review, static analysis, best practices check
Timeframe	December 2025

Document history

VERSION	DATE	CHANGE
1.0	December 2025	Original report issued View original report
1.1	October 2026	Reformatted into the Safe Edges standard template. Findings unchanged

This report was reformatted into the Safe Edges standard template in October 2026. Findings, severities, recommendations and code references are unchanged from the original report, which is preserved unchanged in the [Safe Edges Audit Portfolio on GitHub](#).

CONTENTS

Contents

01	Executive summary	3
02	Scope	4
03	Methodology	5
04	Severity and status	6
05	Findings summary	7
06	Detailed findings	8
	C-01 Missing Loan-Type Validation Allows Unauthorized Loan Filling and Fund Locking	8
	C-02 Liquidation Transfers Entire Remaining Collateral to Lender, Allowing Value Extraction Beyond Outstanding Debt	10
	C-03 Potential Draining of User Funds via fill_lender_request and fill_loan_request	11
	H-01 Lack of Collateralization Threshold Validation During Loan Activation Enables Immediate Liquidation	13
	L-01 Excess Oracle Update Fee Is Not Refunded in pay_and_update_price_feeds	14
	L-02 Missing Validation When Updating protocol_config	15
	I-01 Borrowers Cannot Add Collateral to Avoid Liquidation During Market Volatility	16
	I-02 liquidation_flag_internal Is Never Toggled, Making It Redundant	16
	I-03 Unchecked unwrap() on SRC20::decimals() in check_can_liquidate_based_on_price_ratio_change	17
	Disclaimer	18

Executive summary

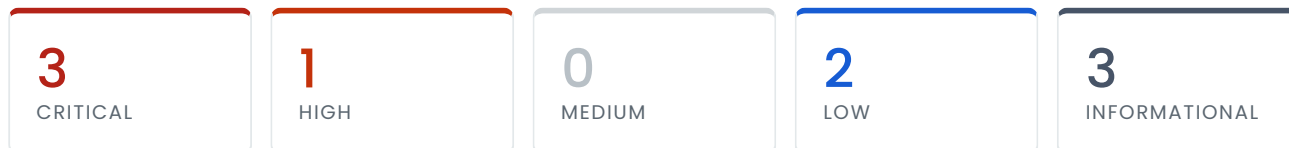
About Axios

Axios is a decentralized Borrow and Lend Isolated Market protocol built on the Fuel network, designed for fixed-rate lending with customizable loan terms. It enables borrowers and lenders to create and fulfill loan requests using specific assets and collateral without shared risk across markets. Axios supports automated liquidation mechanisms powered by Stork price oracles, ensuring loan safety when collateral values fall below defined thresholds. The protocol features time-based and price-based liquidations, automated expirations for unfilled requests, and a transparent fee structure, making it a secure and efficient decentralized money market for fixed-term lending and borrowing.

Summary of the review

Safe Edges reviewed Axios's isolated lending markets on Fuel, written in Sway, in December 2025. Nine issues were identified: three Critical, one High, two Low and three Informational. The Critical issues covered loans being filled without validating the loan type, liquidations that handed the lender all remaining collateral, and a path to drain user funds through the fill functions. Axios fixed seven issues; one Informational issue was acknowledged and one remained unresolved.

Findings by severity



Findings by status

STATUS	ISSUES
Resolved	7
Acknowledged	1
Open	1

Scope

Repository	AxiosFuel/axios-core-contracts
Commit	85c1cb6dfe75ff7d5159cc4916ce4a6bba66e67e
Platform	Fuel
Language	Sway

Only the code at the commit above was reviewed. Changes made after this commit are not covered by this report.

Methodology

Methods recorded for this engagement: Manual code review, static analysis, best practices check.

Manual review

Auditors read the in-scope code line by line, tracing control flow, state changes and trust boundaries against the intended behaviour described by the client.

Threat modelling

We map privileged roles, external dependencies and value flows, then test the assumptions the code makes about each of them.

Functional and automated testing

Where tooling exists for the language and chain, we run static analysis and targeted tests to confirm suspected issues and check edge cases.

Fix review

After the client addresses the findings, we review the updated code and record the final status of each issue.

Severity and status

Severity levels

SEVERITY	DEFINITION
Critical	Direct loss or permanent lock of funds, or full compromise of the protocol, with few or no preconditions. Must be fixed before deployment.
High	Significant loss of funds, broken core functionality or manipulation of critical state under realistic conditions. Should be fixed before deployment.
Medium	Limited loss or degraded functionality, or issues that need specific conditions or privileged access to exploit. Should be fixed.
Low	Minor impact, low likelihood, or deviations from best practice. Fix when practical.
Informational	Code quality, documentation, gas and style observations with no direct security impact.

Status definitions

STATUS	DEFINITION
Resolved	The issue was fixed and the fix was verified.
Partially resolved	The risk was reduced but not fully removed.
Acknowledged	The client accepted the issue without changing the code.
Open	The issue was not addressed at the time of the report.
Not recorded	The original report did not record a resolution status for this issue.

Findings summary

ID	TITLE	SEVERITY	STATUS
C-01	Missing Loan-Type Validation Allows Unauthorized Loan Filling and Fund Locking	Critical	● Resolved
C-02	Liquidation Transfers Entire Remaining Collateral to Lender, Allowing Value Extraction Beyond Outstanding Debt	Critical	● Resolved
C-03	Potential Draining of User Funds via fill_lender_request and fill_loan_request	Critical	● Resolved
H-01	Lack of Collateralization Threshold Validation During Loan Activation Enables Immediate Liquidation	High	● Resolved
L-01	Excess Oracle Update Fee Is Not Refunded in pay_and_update_price_feeds	Low	● Resolved
L-02	Missing Validation When Updating protocol_config	Low	● Resolved
I-01	Borrowers Cannot Add Collateral to Avoid Liquidation During Market Volatility	Informational	● Acknowledged
I-02	liquidation_flag_internal Is Never Toggled, Making It Redundant	Informational	● Open
I-03	Unchecked unwrap() on SRC20::decimals() in check_can_liquidate_based_on_price_ratio_change	Informational	● Resolved

Detailed findings

C-01 Missing Loan-Type Validation Allows Unauthorized Loan Filling and Fund Locking

Severity **Critical** Status ● Resolved

DESCRIPTION

The protocol supports two distinct loan creation flows: `request_loan`, where a borrower creates a loan re-request awaiting a lender, and `offer_loan`, where a lender creates a loan offer awaiting a borrower. However, in both `fill_loan_request` and `fill_lender_request`, the contract fails to validate the loan type or enforce role invariants before assigning the caller as the counterparty. Specifically, `fill_loan_request` unconditionally sets `loan.lender` to the caller address, while `fill_lender_request` unconditionally sets `loan.borrower` to the caller address, without checking whether these fields are already set or whether the loan was created through the corresponding flow.

As a result, any user can fill any pending loan regardless of its intended direction, enabling unauthorized loan filling, griefing, and front-running scenarios in which legitimate borrowers or lenders can be locked out and forced into unintended counterparties, leading to user funds being committed in an unfavorable manner. Specifically, in `fill_loan_request`, the borrower address is neither set during loan creation nor validated during loan filling. As a result, assets are transferred to `Address::zero()` causing the transferred `asset_id` to be irreversibly burned.

```
#[payable, storage(read, write)]
fn fill_lender_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    let mut loan = storage.loans.get(loan_id).read();
    require(loan.status == 0, Error::EInvalidStatus);
    require(
        loan.created_timestamp + get_req_expire_duration() > timestamp(),
        Error::EAlreadyExpired,
    );
    @>> let borrower = get_caller_address();
    loan.borrower = borrower;
    loan.start_timestamp = timestamp();
    loan.status = 2; // magic number 2 is active (ref Enum at interface)
    //...
    let borrower_identity: Identity = get_identity_from_address(borrower);
    transfer(borrower_identity, asset_id, loan.asset_amount);
    log(LoanOfferFilledEvent {
        loan_id,
        borrower: borrower,
        lender: loan.lender,
        asset: loan.asset,
        amount: loan.asset_amount,
        duration: loan.duration,
        liquidation: loan.liquidation.liquidation_flag_internal,
    });
}
```

```
#[payable, storage(read, write)]
fn fill_loan_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    let mut loan = storage.loans.get(loan_id).read();
    require(loan.status == 0, Error::EInvalidStatus);
    require(
        loan.created_timestamp + get_req_expire_duration() > timestamp(),
        Error::EAlreadyExpired,
    );
    @> loan.lender = get_caller_address();
    loan.start_timestamp = timestamp();
    loan.status = 2; // magic number 2 is active (ref Enum at interface)
    //...
    let borrower_identity: Identity = get_identity_from_address(loan.borrower);
    @> transfer(borrower_identity, asset_id, loan.asset_amount);
    log(LoanFilledEvent {
        loan_id,
        borrower: loan.borrower,
        lender: get_caller_address(), //info : should be loan.lender, redundant internal
calls
        asset: loan.asset,
        amount: loan.asset_amount,
        liquidation: loan.liquidation.liquidation_flag_internal,
    });
}
```

IMPACT

An attacker can fill a loan using an incorrect fill function, leaving one counterparty set to the zero address in fill_loan_request. Causing irreversible loss of user funds, and enabling grieving that lock legitimate users out of their loans.

RECOMMENDED MITIGATION

Enforce strict loan-type check before filling:

```
pub enum Error {
+   EBorrowerAlreadySet: (),
+   ELenderAlreadySet: (),
    //...
}
#[payable, storage(read, write)]
fn fill_lender_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    //...
+   require(loan.borrower == Address::zero(), Error::EBorrowerAlreadySet);
+   require(loan.lender != Address::zero(), Error::ELenderAlreadySet);
    let borrower = get_caller_address();
    loan.borrower = borrower;
    loan.start_timestamp = timestamp();
    //...
}
#[payable, storage(read, write)]
fn fill_loan_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    //...
```

```
+ require(loan.borrower != Address::zero(), Error::EBorrowerAlreadySet);
+ require(loan.lender == Address::zero(), Error::ELenderAlreadySet);
loan.lender = get_caller_address();
loan.start_timestamp = timestamp();
loan.status = 2; // magic number 2 is active (ref Enum at interface)
//...
}
```

C-02 Liquidation Transfers Entire Remaining Collateral to Lender, Allowing Value Extraction Beyond Outstanding Debt

Severity Critical Status ● Resolved

DESCRIPTION

The liquidate_loan function distributes liquidation proceeds by deducting the protocol liquidation fee and liquidator incentive from the borrowers collateral, and then transfers the entire remaining collateral balance to the lender, without capping the lenders payout to the outstanding loan amount.

As a result, if the remaining collateral value exceeds the borrowers debt (principal plus interest), the lender captures the surplus value. The borrower receives no refund of excess collateral under any circumstances. Additionally, since the lender can also control the liquidation threshold at loan creation, they can deliberately configure a low liquidation threshold, increasing the likelihood of early liquidation and enabling systematic extraction of borrower collateral beyond debt repayment.

```
#[storage(read, write)]
fn liquidate_loan(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    let mut loan = storage.loans.get(loan_id).read();
    // loan must be active
    require(loan.status == 2, Error::EInvalidStatus);
    let can_loan_be_liquidated = can_liquidate_loan(loan_id);
    if (can_loan_be_liquidated) {
        let protocol_fee = (loan.collateral_amount * get_protocol_liq_fee()) / 10000;
        let liquidator_amount = (loan.collateral_amount * get_liquidator_fee()) / 10000;
        let lender_amount = loan.collateral_amount - liquidator_amount - protocol_fee;
        loan.status = 4;
        storage.loans.insert(loan_id, loan);
        let collateral_asset_id: AssetId = get_asset_id_from_b256(loan.collateral);
        let lender_identity: Identity = get_identity_from_address(loan.lender);
    @>> transfer(lender_identity, collateral_asset_id, lender_amount);
        //....
    }
}
```

Potential Scenario:

1. A borrower deposits 20 ETH as collateral, valued at \$2,500 per ETH, for a total collateral value of \$50,000. 2.The borrower takes a loan of \$30,000 USDT with a liquidation threshold of 70%.
- 3.Due to market volatility, the price of ETH drops to \$2,000 per ETH, reducing the collateral value to \$40,000 and bringing the position below the liquidation threshold.

4.The protocol defines both the liquidator fee and the protocol liquidation fee as 10% each (1,000 bps).

5.A liquidator calls `liquidate_loan`, resulting in:

\$4,000 paid to the liquidator as an `liquidator_fee`

\$4,000 paid to the protocol as a `protocol_liquidation_fee`

6.After deducting fees, \$32,000 worth of ETH remains.

7.Under the current implementation, the entire remaining \$32,000 of collateral is transferred to the lender, despite the outstanding loan amount being only \$30,000 (without interest).

8.As a result, the lender captures an additional \$2,000 beyond the loan principal, while the borrower receives no refund of excess collateral, effectively allowing the lender to profit more from liquidation than from loan interest.

IMPACT

Lenders can extract value beyond the outstanding loan amount, resulting in borrowers permanently losing excess collateral. This creates unfair, lender-favored incentives and enables systematic borrower losses without any technical exploitation.

RECOMMENDED MITIGATION

Cap the lenders liquidation payout to the outstanding loan amount (principal plus accrued interest, if applicable). After paying the liquidator fee and protocol liquidation fee, any remaining collateral surplus must be refunded to the borrower.

C-03 Potential Draining of User Funds via `fill_lender_request` and `fill_loan_request`

Severity Critical Status ● Resolved

DESCRIPTION

The protocol supports two distinct loan creation flows:

- Lender-offered loans, where a lender escrows the loan asset and waits for a borrower.
- Borrower-requested loans, where a borrower escrows collateral and waits for a lender.

However, as detailed in C-01, the protocol does not explicitly encode or validate the loan type during the loan-filling phase. As a result, both `fill_lender_request` and `fill_loan_request` can be invoked on loans created through the opposite flow, without enforcing role or escrow invariants.

This design flaw allows the following unsafe behaviors:

- Assets held by the contract for one loan may be used to satisfy a different loan.
- Lender or borrower addresses can be overwritten incorrectly.
- Transfers may be executed to the zero address.
- Escrowed assets or collateral can become permanently unrecoverable.

1. `fill_lender_request` Called on Borrower-Requested Loans

Even though such loans do not have lender escrow backing them. If the contract currently holds assets from unrelated lender-offered loans, these funds are used to satisfy the transfer, even though they were escrowed for a different loan.

Since the loans lender field was never initialized, it remains the zero address. Upon loan expiry or liquidation, the remaining collateral is transferred to the lender address, resulting in the collateral being burned. This constitutes a grieving attack that permanently destroys user funds.

2. fill_loan_request Called on Lender-Offered Loans

As such loans were never associated with a borrower. The original lender can no longer cancel the loan or reclaim the escrowed funds, as they are no longer recognized as the lender. So when the loan expires, the protocol transfers collateral from its global balance to the lender, which belongs to a different loan. This enables a potential net profit funds draining attack.

Potential Attack Scenario:

1.Alice creates a lender-offered loan, escrowing 30,000 USDC, expecting ETH collateral worth 50,000 USD. 2.An attacker calls fill_loan_request(aliceLoanId) on this loan.

3.The attacker transfers 30,000 USDC as required and is set as the lender.

4.The contract transfers 30,000 USDC to loan.borrower, which is the zero address.

5.Alice attempts to cancel the loan but fails because she is no longer recognized as the lender. 6.Alices escrowed USDC is permanently lost.

7.Upon loan expiry, a liquidator calls liquidate_loan(aliceLoanId). From the \$50,000 worth of collateral, a 10% liquidator fee (\$5,000) and a 10% protocol liquidation fee (\$5,000) are deducted.

8.The remaining \$40,000 worth of collateral is transferred to the attacker.

9.As a result, the attacker makes a profit of 10,000 USD at the expense of Alice's 30,000 USD loss.

IMPACT

This issue can potentially leads to permanent loss of user's funds, grieving of legitimate lenders and borrowers and protocol accounting inconsistency.

RECOMMENDED MITIGATION

Apply the same strict loan-type checks described in [C-01](#c-01--missing-loan-type-validation-allows-unauthorized-loan-filling-and-fund-locking), ensuring that fill_lender_request and fill_loan_request can only be executed on loans created through their corresponding flows.

H-01 Lack of Collateralization Threshold Validation During Loan Activation Enables Immediate Liquidation

Severity **High** Status ● Resolved

DESCRIPTION

The protocol does not validate the collateralization ratio when a loan transitions into the Active state. Specifically, in both `fill_lender_request` and `fill_loan_request`, the contract activates the loan without verifying that the collateral value satisfies the configured liquidation threshold based on current oracle prices. Since liquidation eligibility is determined using live oracle prices, it is possible for a loan to be activated while already below the liquidation threshold due to price movements occurring between loan creation and loan fill. In such cases, the loan becomes immediately liquidatable upon activation.

This creates a critical inconsistency where a loan can enter the system in an already insolvent state.

```
#[payable, storage(read, write)]
fn fill_lender_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    //...
    require(asset_id == loan.collateral, Error::EInvalidCollateral);
    require(
        amount == loan.collateral_amount,
        Error::EInvalidCollateralAmount,
    );
    let asset_id: AssetId = get_asset_id_from_b256(loan.asset);
    let borrower_identity: Identity = get_identity_from_address(borrower);
    @>> transfer(borrower_identity, asset_id, loan.asset_amount);
}

#[payable, storage(read, write)]
fn fill_loan_request(loan_id: u64) {
    require(!is_protocol_paused(), Error::EProtocolPaused);
    //...
    require(asset_id == loan.asset, Error::EInvalidAsset);
    require(amount == loan.asset_amount, Error::EInvalidAssetAmount);
    let asset_id: AssetId = get_asset_id_from_b256(loan.asset);
    let borrower_identity: Identity = get_identity_from_address(loan.borrower);
    @>> transfer(borrower_identity, asset_id, loan.asset_amount);
}
```

IMPACT

A borrowers loan can be liquidated immediately after activation causing instant loss of collateral without the borrower ever having a viable borrowing period. This exposes borrowers to unfair and unavoidable losses during periods of market volatility and enables opportunistic liquidation strategies.

RECOMMENDED MITIGATION

Before transitioning a loan from Pending to the Active state the protocol should compute the current collateralization ration and ensure it is above the liquidation threshold.

L-01 Excess Oracle Update Fee Is Not Refunded in `pay_and_update_price_feeds`

Severity **Low** Status ● Resolved

DESCRIPTION

The `pay_and_update_price_feeds` function allows users to update Stork oracle price feeds by paying the required update fee. However, the function only enforces that the caller provides at least the required fee (`msg_amount() >= stork_update_fee`) and does not refund any excess amount supplied.

During execution, only `stork_update_fee` is forwarded to the Stork oracle contract, while any surplus funds remain locked in the protocol contract. Since the current implementation does not include any withdrawal or recovery mechanism for arbitrary balances, these excess funds become permanently inaccessible.

IMPACT

If a user accidentally overpays the oracle update fee, the excess amount will be irreversibly locked in the contract.

RECOMMENDED MITIGATION

Consider enforcing exact fee payment approach:

```
#[payable, storage(read)]
fn pay_and_update_price_feeds(update_data: Vec<TemporalNumericValueInput>) {
    let stork_contract_id = storage.stork_contract.read();
    require(
        stork_contract_id != ContractId::zero(),
        Error::EOracleNotSet,
    );
    let stork_oracle_dispatcher = abi(Stork, stork_contract_id.bits());
    let stork_update_fee = stork_oracle_dispatcher.get_update_fee_v1(update_data);
    require(
        - msg_amount() >= stork_update_fee,
        + msg_amount() == stork_update_fee,
        Error::ENotEnoughForOracleUpdate,
    );
    stork_oracle_dispatcher
        .update_temporal_numeric_values_v1 {
            asset_id: msg_asset_id().bits(),
            coins: stork_update_fee,
        }(update_data);
}
```

L-02 Missing Validation When Updating protocol_config

Severity **Low** Status ● Resolved

DESCRIPTION

The `update_protocol_config` function allows the protocol admin to overwrite the entire `protocol_config` struct without enforcing any validation or invariants on its fields.

```
fn update_protocol_config(config: ProtocolConfig) {
    only_protocol_admin();
    storage.protocol_config.write(config);
}
```

Because `protocol_config` governs core economic and safety parameters of the system, the absence of validation allows invalid or dangerous values to be set, which can lead to protocol malfunction, denial of service, or direct loss of user funds.

IMPACT

Misconfigured protocol parameters can place the system into unsafe or non-functional states, including permanent fee loss, broken economic incentives, DoS of core loan flows when `time_request_loan_expires` is zero, and immediate borrowers loss when `min_loan_duration` is zero.

RECOMMENDED MITIGATION

Enforce strict validation in `update_protocol_config` before writing the configuration to storage. The checks below use predefined default values as minimum safety thresholds.

```
pub enum Error {
+   EInvalidProtocolConfig: (),
    //...
}

fn update_protocol_config(config: ProtocolConfig) {
    only_protocol_admin();
+   require(config.protocol_fee_receiver != Address::zero(), Error::EInvalidProtocolConfig);
+   require(config.liquidator_fee > 0, Error::EInvalidProtocolConfig);
+   require(config.protocol_fee + config.liquidator_fee + config.protocol_liquidation_fee <
10_000, Error::EInvalidProtocolConfig);
+   require(config.time_request_loan_expires > 0, Error::EInvalidProtocolConfig);
+   require(config.oracle_max_stale >= 30, Error::EInvalidProtocolConfig);
+   require(config.min_loan_duration >= 600, Error::EInvalidProtocolConfig);
    storage.protocol_config.write(config);
}
```

I-01 Borrowers Cannot Add Collateral to Avoid Liquidation During Market Volatility

Severity **Informational** Status ● Acknowledged

DESCRIPTION

The protocol does not provide any mechanism for borrowers to add additional collateral to an active loan. Once a loan is live, the borrower cannot increase their collateralization ratio, even if the loan becomes close to liquidation due to sudden market volatility or oracle price fluctuations.

In scenarios where the borrower expects market conditions to recover before loan expiration, they may be willing to deposit additional collateral to maintain a safe collateralization level. However, the current design forces borrowers to either fully repay the loan or risk liquidation, removing any ability to actively manage liquidation risk.

RECOMMENDED MITIGATION

Consider introducing a mechanism that allows borrowers to add collateral to active loans.

I-02 `liquidation_flag_internal` Is Never Toggled, Making It Redundant

Severity **Informational** Status ● Open

DESCRIPTION

The contract uses the `liquidation_flag_internal` flag to determine whether oracle-based liquidation checks should be applied to a loan. However, this flag is always initialized to true during loan creation and is never updated thereafter.

There is no function—admin-controlled or otherwise—that allows this flag to be toggled post-creation. As a result, all loans are permanently subject to oracle-based liquidation, and the flag does not meaningfully influence protocol behavior.

If the intended design was to allow the protocol to selectively exempt certain loans from oracle-based liquidation (e.g., special agreements, whitelisted counterparties, or off-chain-managed risk), the current implementation does not support this functionality.

IMPACT

The flag represents incomplete or misleading functionality.

RECOMMENDED MITIGATION

If oracle-based liquidation is not intended to apply to all loans, introduce an admin-controlled mechanism to toggle `liquidation_flag_internal` on a per-loan basis, enabling selective exemption from oracle-based liquidation.

I-03 Unchecked unwrap() on SRC20::decimals() in check_can_liquidate_based_on_price_ratio_change

Severity Informational Status ● Resolved

DESCRIPTION

The contract retrieves token decimal information using decimals() from the SRC20 interface and immediately calls unwrap() on the returned value. If the configured asset's decimals() returns None, the call will panic and revert.

While oracle configurations are restricted to the protocol admin, misconfiguration or incorrect asset registration (less likely) can still lead to unexpected reverts.

Disclaimer

This report is based on the code at the commit listed in the Scope section and on information the client provided during the review. It reflects Safe Edges' assessment at the time of the review and is not a guarantee that the code is free of vulnerabilities.

Changes made after the reviewed commit, deployment configuration, off-chain components and third-party dependencies are outside the scope of this report unless stated otherwise.

Security is ongoing. We recommend continued testing, monitoring and bug bounty coverage alongside any audit.

This report is not investment advice and is not an endorsement of the project, its team or any token.



Safe Edges is a Web3 security firm and official security partner for the Fuel, ENI and XI EcoChain ecosystems. We work across smart contract audits, Rust and WASM reviews, ZK circuit review, AI and MCP security, and penetration testing.

200+

PROJECTS AUDITED

\$4B+

TVL SECURED

0

POST-AUDIT HACKS