



SMART CONTRACT SECURITY AUDIT

Yuchifun

Bonding Curve Contracts

CLIENT

Yuchifun

PLATFORM

EVM

LANGUAGE

Solidity

REVIEW

October 2025

DOCUMENT

Document information

Project	Yuchifun, Bonding Curve Contracts
Client	Yuchifun
Platform	EVM
Language	Solidity
Repository	Not recorded in the original report
Commit	Not recorded in the original report
Methods	Not recorded in the original report
Report date	October 2025 (original says 2025; date taken from the PDF)

Document history

VERSION	DATE	CHANGE
1.0	October 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
	M-01 Missing Check for Existing Pair in FFactory::createPair	8
	M-02 Lack of Slippage Protection in Bonding::sell and Bonding::buy Functions	9
	M-03 Missing Maximum Value Restrictions	10
	M-04 Race Condition Vulnerability in FERC20::approve Function	10
	L-01 Missing Token Address Validation in FFactory::createPair	11
	L-02 Missing Input Validation in Multiple Functions	12
	L-03 Stuck ETH in Bonding::buy Function	12
	L-04 Incorrect usage of AccessControl in upgradeable contracts	13
	I-01 Use Ownable2StepUpgradeable for Ownership Transfers	13
	G-01 Validation of Pair Existence for Gas Optimization	14
	Disclaimer	15

Executive summary

About Yuchifun

The reviewed contracts run a bonding-curve token launch. FFactory creates trading pairs, FRouter executes trades, FERC20 is the launched token, and the Bonding contract lets users buy and sell launched tokens against an asset token.

Summary of the review

Safe Edges reviewed Yuchifun's Solidity bonding-curve contracts (FFactory, FRouter, FERC20 and Bonding) in 2025. Ten issues were identified: four Medium, four Low and two Informational, one of which is a gas optimisation. The Medium issues covered duplicate trading pairs, buys and sells without slippage protection, admin parameters without upper bounds, and the ERC-20 approve race condition. The original report did not record resolution statuses.

Findings by severity



Findings by status

STATUS	ISSUES
● Not recorded	10

Scope

Repository	Not recorded in the original report
------------	-------------------------------------

Commit	Not recorded in the original report
--------	-------------------------------------

Platform	EVM
----------	-----

Language	Solidity
----------	----------

IN-SCOPE FILES

- FFactory
- FRouter
- FERC20
- Bonding

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: Not recorded in the original report.

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
M-01	Missing Check for Existing Pair in <code>FFactory::createPair</code>	Medium	● Not recorded
M-02	Lack of Slippage Protection in <code>Bonding::sell</code> and <code>Bonding::buy</code> Functions	Medium	● Not recorded
M-03	Missing Maximum Value Restrictions	Medium	● Not recorded
M-04	Race Condition Vulnerability in <code>FERC20::approve</code> Function	Medium	● Not recorded
L-01	Missing Token Address Validation in <code>FFactory::createPair</code>	Low	● Not recorded
L-02	Missing Input Validation in Multiple Functions	Low	● Not recorded
L-03	Stuck ETH in <code>Bonding::buy</code> Function	Low	● Not recorded
L-04	Incorrect usage of <code>AccessControl</code> in upgradeable contracts	Low	● Not recorded
I-01	Use <code>Ownable2StepUpgradeable</code> for Ownership Transfers	Informational	● Not recorded
G-01	Validation of Pair Existence for Gas Optimization	Informational	● Not recorded

Detailed findings

M-01 Missing Check for Existing Pair in FFactory::createPair

Severity **Medium** Status ● Not recorded

DESCRIPTION

The `createPair` function does not check if a trading pair already exists before creating a new one. This could lead to conflicts when multiple pairs of the same tokens are created.

IMPACT

Potential conflicts in identifying which pair address to use in further operations.

RECOMMENDATION

Implement a check to ensure that a trading pair does not already exist before creating a new pair.

```
function createPair(address tokenA, address tokenB) external onlyRole(CREATOR_ROLE) nonReentrant
returns (address) {
    // Check if pair already exists
    require(_pair[tokenA][tokenB] == address(0), "Pair already exists");
    // Proceed with creating pair
    address pair = _createPair(tokenA, tokenB);
    return pair;
}
```

M-02 Lack of Slippage Protection in `Bonding::sell` and `Bonding::buy` FunctionsSeverity **Medium** Status ● Not recorded

DESCRIPTION

The `sell` and `buy` functions in the `Bonding` contract enable users to trade base tokens for asset tokens. However, these functions lack slippage protection mechanisms, which can result in users paying excessive prices for assets due to market fluctuations. This exposes the contract to potential "sandwich attacks."

SCENARIO EXAMPLE

Assume a pool where the ratio of `memeToken` to `assetToken` is 10:1, meaning the asset token is 10 times more valuable. If a user (Alice) attempts to buy `memeToken`, an attacker (Bob) can exploit the lack of slippage protection by executing a sandwich attack. Bob front-runs Alice's transaction, inflating the price of `memeToken`, and then back-runs Alice's transaction, selling at a higher price, causing Alice to pay significantly more than expected.

IMPACT

- Users can suffer significant financial losses due to slippage.
- Attackers can exploit the situation to profit at the user's expense.

PROOF OF CONCEPT

The absence of a slippage control mechanism allows attackers to manipulate prices within the trading pool.

RECOMMENDATION

Implement the `minAmountOut` parameter in the `buy` and `sell` functions to provide slippage protection, ensuring that users are not subjected to unexpected price changes.

```
function buy(uint256 amountIn, address tokenAddress) public payable returns (bool) {  
    // Existing logic here...  
    // Slippage protection - use minAmountOut to prevent excessive slippage  
    uint256 minAmountOut = calculateMinAmountOut(amountIn, tokenAddress);  
    require(amountOut >= minAmountOut, "Slippage too high");  
    // Continue function logic...  
}
```

M-03 Missing Maximum Value Restrictions

Severity **Medium** Status ● Not recorded

DESCRIPTION

Functions like `FERC20::updateMaxTx` and `Bonding::setFee` lack restrictions on maximum values for critical parameters, which could result in the application of unreasonable values.

IMPACT

Lack of safeguards could lead to the setting of excessive transaction limits or fees, potentially destabilizing the token economy.

RECOMMENDATION

Implement checks to restrict the maximum allowable values for sensitive parameters like `maxTx` and `fee`.

```
function updateMaxTx(uint256 newMaxTx) external onlyRole(ADMIN_ROLE) {
    require(newMaxTx <= MAX_TX_LIMIT, "Exceeds maximum transaction limit");
    maxTx = newMaxTx;
}
```

M-04 Race Condition Vulnerability in `FERC20::approve` Function

Severity **Medium** Status ● Not recorded

DESCRIPTION

The `approve` function in the `FERC20` contract is susceptible to a well-known race condition. If a user (Alice) approves a certain allowance for another user (Bob) and then attempts to reduce or modify this allowance, Bob could front-run the transaction, using the previously approved allowance before Alice's update is applied and after that can also redeem the newly approved allowance.

IMPACT

Malicious users could exploit this vulnerability to spend more tokens than the user intended.

RECOMMENDATION

Use the `increaseAllowance` or `decreaseAllowance` functions, which modify allowances atomically to prevent front-running. Alternatively, reset the allowance to 0 before updating to a new value.

```
function approve(address spender, uint256 amount) public returns (bool) {
    uint256 currentAllowance = allowance[msg.sender][spender];
    // Mitigate race condition: reset allowance to 0 first before changing to new amount
    if (currentAllowance != 0) {
        _approve(msg.sender, spender, 0);
    }
    _approve(msg.sender, spender, amount);
    return true;
}
```

L-01 Missing Token Address Validation in FFactory::createPair

Severity **Low** Status ● Not recorded

DESCRIPTION

The `createPair` function does not check whether the two provided token addresses are different. This could allow the creation of a pool where both tokens are the same address, making the pool ineffective.

IMPACT

The creation of a token pair where both tokens are identical undermines the intended purpose of an Automated Market Maker (AMM).

RECOMMENDATION

Implement a validation check to ensure that tokenA and tokenB are not the same address.

```
function createPair(address tokenA, address tokenB) external onlyRole(CREATOR_ROLE) nonReentrant
returns (address) {
    // Check if tokens are the same
    require(tokenA != tokenB, "Tokens should be different");
    // Proceed with creating the pair
    address pair = _createPair(tokenA, tokenB);
    return pair;
}
```

L-02 Missing Input Validation in Multiple Functions

Severity **Low** Status ● Not recorded

DESCRIPTION

Several functions in the protocol lack input validation, especially checks for zero values. This could lead to unintended behaviors and vulnerabilities.

IMPACT

Functions could execute with invalid or unintentional inputs, potentially causing issues or unexpected behavior.

RECOMMENDATION

Add necessary checks for zero inputs across multiple functions, such as `FFactory::initialize` and `FRouter::sell`.

```
function initialize(address taxVault_, uint256 buyTax_, uint256 sellTax_) external initializer {
    // Ensure non-zero taxVault address
    require(taxVault_ != address(0), "Zero taxVault address");
    taxVault = taxVault_;
    buyTax = buyTax_;
    sellTax = sellTax_;
}

function sell(uint256 amountIn, address tokenAddress, address to) public nonReentrant
onlyRole(EXECUTOR_ROLE) returns (uint256, uint256) {
    // Ensure no zero addresses or amounts
    require(tokenAddress != address(0), "Zero addresses are not allowed");
    require(to != address(0), "Zero addresses are not allowed");
    require(amountIn != 0, "Zero Amount In");
    // Logic for sell
}
```

L-03 Stuck ETH in Bonding::buy Function

Severity **Low** Status ● Not recorded

DESCRIPTION

The `Bonding::buy` function is defined as `payable`, but any ETH sent with the transaction is neither tracked nor transferred. This results in the ETH being stuck in the contract indefinitely.

IMPACT

ETH sent along with the transaction could be lost or stuck in the contract, leading to user dissatisfaction and possible financial losses.

RECOMMENDATION

Remove the `payable` modifier from the function or implement a mechanism to transfer incoming ETH to a designated address.

```
function buy(uint256 amountIn, address tokenAddress) public payable returns (bool) {
    // Logic for purchasing asset token here
    // If ETH is sent, transfer it out to a defined address
    if (msg.value > 0) {
        payable(owner()).transfer(msg.value); // Transfer ETH to owner or another address
    }
    return true;
}
```

L-04 Incorrect usage of `AccessControl` in upgradeable contracts

Severity Low Status ● Not recorded

DESCRIPTION

In the current contract, `AccessControl` (non-upgradeable version) is used in an upgradeable contract. Using the non-upgradeable variant of `AccessControl` can result in issues with uninitialized state, potential security vulnerabilities, and failure to follow upgradeability patterns. Specifically, upgradeable contracts should utilize `AccessControlUpgradeable` from OpenZeppelin to ensure proper initialization and compatibility with upgradeable contract patterns.

```
import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol"; import "
```

I-01 Use `Ownable2StepUpgradeable` for Ownership Transfers

Severity Informational Status ● Not recorded

DESCRIPTION

To mitigate the risk of ownership issues, it's recommended to use the `Ownable2StepUpgradeable` contract for ownership transfers.

IMPACT

Prevents scenarios where ownership transfers could fail due to the next owner being non-existent or not accepting ownership.

RECOMMENDATION

Replace current ownership logic with `Ownable2StepUpgradeable` for better security.

```
import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";
contract Bonding is Ownable2StepUpgradeable {
    // Your contract logic here
}
```

G-01 Validation of Pair Existence for Gas Optimization

Severity

Informational

Status

● Not recorded

Category

Gas

DESCRIPTION

When deriving the pair address in the `Bonding.sol` and `FRouter.sol` contracts, the existence of the pair address is not checked before performing operations, which can result in unnecessary gas consumption and failed transactions.

IMPACT

Increased gas costs and failed operations when the pair does not exist.

RECOMMENDATION

Perform a validation check to ensure the pair exists before proceeding with operations.

```
address pairAddress = factory.getPair(token, assetToken);
if (pairAddress == address(0)) {
    revert ZeroPairAddress(); // Revert if pair does not exist
}
```

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