QuillAudits

Audit Report, August, 2024

For

FRINGE

Table of Content

Executive Summary

Project Name	Fringe Finance
Overview	Fringe finance is a borrowing lending platform with new features such as partial liquidations.
Timeline	5th July 2024 - 9th August 2024
Updated Code Received	23rd August 2024
Second Review	26th August 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyze the Fringe Finance codebase for quality, security, and correctness.
Source Code	https://github.com/fringe-finance/primary-smart-contracts/tree/cc48efe2868c017e0f4b62f5278be081d2baaa9a
Contracts In-Scope	All list of changed contracts for Fringe V2.5: Updated: BlendingToken.sol PriceProvider.sol ChainlinkPriceProvider.sol PythPriceProvider.sol wstETHPriceProvider.sol PriceProviderAggregator.sol PriceProviderAggregatorPyth.sol PrimaryLendingPlatformV2Core.sol PrimaryLendingPlatformAtomicRepaymentCore.sol PrimaryLendingPlatformLeverageCore.sol PrimaryLendingPlatformLiquidationCore.sol PrimaryLendingPlatformWrappedTokenGatewayCore.sol PrimaryLendingPlatformV2Zksync.sol PrimaryLendingPlatformAtomicRepaymentZksync.sol

Executive Summary

PrimaryLendingPlatformLeverageZksync.sol
PrimaryLendingPlatformLiquidationZksync.sol
PrimaryLendingPlatformWrappedTokenGatewayZksync.sol
PrimaryLendingPlatformModeratorCore.sol
Added:
LPPriceProvider.sol
UniswapV3PriceProvider.sol
ERC4626PriceProvider.sol
PriceOracle.sol
Asset.sol
Errors.sol

Branch	Main
Contracts out of Scope	In-scope contract has been audited by QuillAudits. However, these contracts inherit functionality from out-of-scope Smart contracts that were not audited. Vulnerabilities in unaudited contracts could impact in-scope Smart Contracts functionality. QuillAudits is not responsible for such vulnerabilities. **Below are Out of Scope Contracts:** • contracts/interfaces/* • OpenZeppelin contracts (Initializable.sol, ...)
Fixed In	https://github.com/fringe-finance/primary-smart-contracts/commits/feature/plpv2.5-audit-fix/contracts **Commit hash: 68a3531aa6c818bd4c0b9405bcc23e55671ab3ab**

Number of Issues per Severity

4
Issues Found

🟥 High	🟨 Medium
🟩 Low	🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	1	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	0	0	0

Checked Vulnerabilities

✓ Reentrancy	✓ Transfer forwards all gas
✓ Timestamp Dependence	✓ ERC20 API violation
✓ Gas Limit and Loops	✓ Compiler version not fixed
✓ DoS with Block Gas Limit	✓ Redundant fallback function
✓ Transaction-Ordering Dependence	✓ Send instead of transfer
✓ Use of tx.origin	✓ Style guide violation
✓ Exception disorder	✓ Unchecked external call
✓ Gasless send	✓ Unchecked math
✓ Balance equality	✓ Unsafe type inference
✓ Byte array	✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC's standards.
  Efficient use of gas.
  Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Manual Review, Slither, Hardhat.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

A1. Inflation attack

Path

BErc20.sol

Function

getCashPrior()

Description

Defi lending protocols such as fringe finance lets you deposit $100 worth of token A to borrow 50-80% amount of token B. Here to manipulate the price of the tokens oracles can be manipulated to inflate the price of the token. The issue lies when the collateral is withdrawn through redeem function. There is possibility that fToken may give value close to full value but not complete value. Ie, it might return 1.9999995

But not 2. This might result in rounding the value down resulting in only returning 1.

The calculation of the exchangeRate, as previously introduced, involves getCashPrior(), which refers to the amount of underlying balance owned by the fToken contract. By directly transferring underlying tokens into the contract (without mint, just transferring), the hacker can manipulate the exchangeRate.

Remediation

Ensure that markets are never empty by minting small amount of fToken at the time of market creation, preventing the rounding error from being used maliciously. A possible approach is using uniswapV2 implementation that locks small amount of tokens permanently.

Status

Resolved

Fringe Finance Team's Comment

We will be the first to supply fToken right after fToken is deployed and transfer the corresponding amount of fToken to 0x00 address. The meaning of this action is:

- Make sure that the total supply of fToken never goes to zero
- Permanently lock the number of certain number of fTokens to 0x00 address at the first supply of capital assets.

Also the new oracle price model has been implemented.

Medium Severity Issues

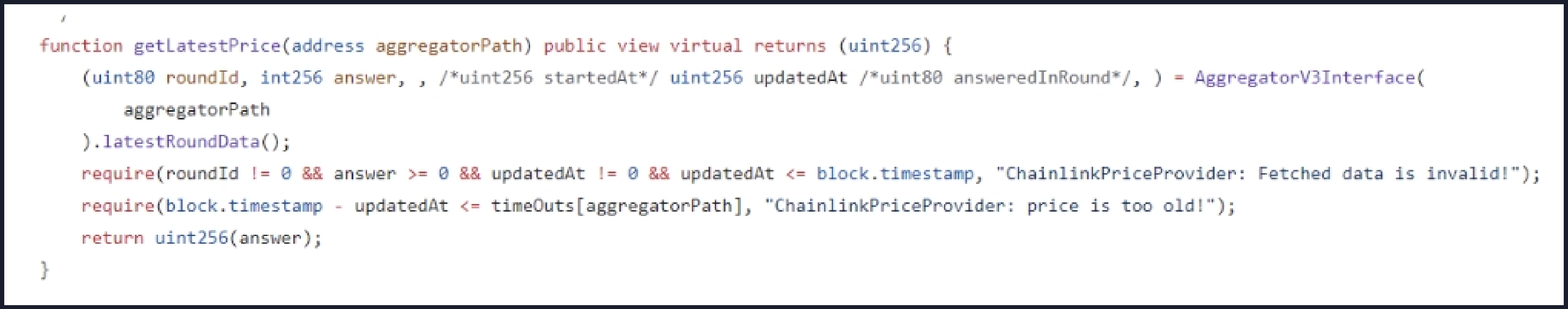
B1. ChainlinkPriceProvider does not implement check against min price

Path

ChainlinkPriceProvider.sol

Function

getLastprice()



```
function getLatestPrice(address aggregatorPath) public view virtual returns (uint256) {
    (uint80 roundId, int256 answer, , /*uint256 startedAt*/ uint256 updatedAt /*uint80 answeredInRound*/, ) = AggregatorV3Interface(
        aggregatorPath
    ).latestRoundData();
    require(roundId != 0 && answer >= 0 && updatedAt != 0 && updatedAt <= block.timestamp, "ChainlinkPriceProvider: Fetched data is invalid!");
    require(block.timestamp - updatedAt <= timeOuts[aggregatorPath], "ChainlinkPriceProvider: price is too old!");
    return uint256(answer);
}
```

Description

ChainlinkPriceProvider have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value (i.e. LUNA crash) the price of the oracle will continue to return the minPrice instead of the actual price of the asset. This would allow users to continue borrowing with the asset but at the wrong price. This is exactly what happened to **Venus on BSC when LUNA imploded**.

ChainlinkPriceProvider have minPrice and maxPrice circuit breakers built into them. This means that if the price of the asset drops below the minPrice, the protocol will continue to value the token at minPrice instead of its actual value. This will allow users to take out huge amounts of bad debt and bankrupt the protocol.

Example: TokenA has a minPrice of $1. The price of TokenA drops to $0.10. The aggregator still returns $1 allowing the user to borrow against TokenA as if it is $1 which is 10x its actual value.

Recommendation

ChainlinkPriceProvider should check the returned answer against the minPrice/maxPrice and revert if the answer is outside of the bounds:

```
(uint80 roundId, int256 answer, , /*uint256 startedAt*/ uint256 updatedAt /*uint80
answeredInRound*/, ) = AggregatorV3Interface(aggregatorPath
        ).latestRoundData();

++   if (answer >= maxPrice or answer <= minPrice) revert();
```

Status

Acknowledged

Fringe Finance Team's Comment

It would make more sense to have an off-chain backend system to monitor the price from Chainlink. If the price is incorrect, we would intervene to pause the oracle.

Low Severity Issues

3. Add _disableInitializer() in implementation contract's constructor [Common Issue]

Description

In proxy contracts where there is initialize() function without access control can be front run by attackers. From OZ blog:

At the time of deploying the contract, the protocol owner initializes the proxy contract, and the protocol owner becomes the owner of the contract. However, an attacker can call initialize function on the implementation contract and become the owner of the implementation contract. In this case, the attacker becomes the owner of the implementation contract. If the implementation contract's delegatecall is executed, it could introduce a critical vulnerability. Specifically, the attacker could use delegatecall to execute code from the implementation contract that triggers attack.sol to self-destruct. As a result, all calls made by the proxy contract would fail.

Remediation

To remediate the issue please make sure to add

```
    constructor() {
    _disableInitializer();
};
```

Access control such as owner variable can be added.
Also on other hand make sure to initialize the function while deploying.

Status

Acknowledged

Fringe Finance Team's Comment

At contract deployment time, the protocol deployer will call the initialize() function to become the admin and moderator of both the proxy contract and the implementation contract. Initializing the implementation will prevent any malicious attacker from calling the initialize() function and taking control of the implementation contract or self-destructing itself.

Informational Issues

1. “OnlyAdmin” modifier not used in many of the contracts

Status

Acknowledged

Functional Tests Cases

Chainlink:

- ✓ Price is correct

Liquidation:

- ✓ Can liquidate full amount
- ✓ Can liquidate partial amount
- ✓ Can liquidate amount and get incentives
- ✓ Liquidation is working when HF is less than 1
- ✓ Liquidation reverts when lending amount is greater than maxLA

Repayments:

- ✓ Should only use bytes in updateData
- ✓ Should revert when amountSold > collateral amount
- ✓ Should revert when HF < 1, isLeveragePosition is false

Wrapped Token:

- ✓ Should revert when allowance of WETH of user for contract is less than lending Token amount
- ✓ Should revert when availableToBorrow is 0 and loanBody is 0
- ✓ Should revert when totalBorrowPerCollateral is greater than borrowLimitPerCollateral
- ✓ Should be able to borrow some USDC tokens when isLeveragePosition is true
- ✓ Should revert if collateral factor is greater than 100
- ✓ Sould not allow maximum loss

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Fringe Finance codebase. We performed our audit according to the procedure described above.

Issues of High, Medium, Low and Informational severity were found, suggestions and best practice are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Fringe Finance smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Fringe Finance smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Fringe Finance to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report August, 2024

For

FRINGE

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com