SMART CONTRACT AUDIT REPORT

for

Extra Finance

Prepared By: Xiaomi Huang

PeckShield

November 8, 2024

Document Properties

Client	Extra Finance
Title	Smart Contract Audit Report
Target	Extra Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 8, 2024	Xuxian Jiang	Final Release
1.0-rc	November 5, 2024	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Extra Finance` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Extra Finance

`Extra Finance` protocol forks from `AaveV3` – the popular decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. `Extra Finance` maintains the same core business logic, but reconstructs with new staking component and enhanced debt payment. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Extra Finance

Item	Description
Name	Extra Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 8, 2024

In the following, we show the Git repositories of reviewed files and the commit hash value used in this audit. Note that Extra Finance assumes a trusted price oracle with timely market price feeds for supported assets.

- https://github.com/ExtraFi/extra-x-contracts.git (16828b4)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/ExtraFi/extra-x-contracts.git (360fd57)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Extra Finance` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	3	■■■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Extra Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Rate Adjustment Upon Pool Interest Rate Strategy Change	Business Logic	Resolved
PVE-002	Low	Improved Asset Addition Logic in ConfiguratorLogic	Business Logic	Resolved
PVE-003	Medium	Incorrect Reward Calculation in ATokenRewardsReDistributionManager	Business Logic	Resolved
PVE-004	Low	Lack of MAX_ACCOUNTS_PER_USER Enforcement in ExtraXAccountFactory	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Timely Rate Adjustment Upon Pool Interest Rate Strategy Change

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pool
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The Extra Finance protocol allows the governance to dynamically configure the interest rate strategy for current reserves. The supported interest rate strategy implements the calculation of the interest rates depending on the reserve state. While reviewing current configuration logic, we notice the update of the interest rate strategy warrants the need of refreshing the latest stable borrow rate, the latest variable borrow rate, as well as the latest liquidity rate.

To elaborate, we show below the setReserveInterestRateStrategyAddress() function. It implements a rather straightforward logic in validating and applying the new interestRateStrategyAddress contract. It comes to our attention that the internal accounting for various rates is not timely refreshed to make it immediately effective. As a result, even if the interest rate strategy is already updated, current rates are not updated yet. In other words, the latest stable/variable borrow rate and the latest liquidity rate are still based on the replaced interest rate strategy.

```
function setReserveInterestRateStrategyAddress(address asset, address
    rateStrategyAddress)
  external
  virtual
  override
  onlyPoolConfigurator
{
  require(asset != address(0), Errors.ZERO_ADDRESS_NOT_VALID);
```

```
    require(_reserves[asset].id != 0  _reservesList[0] == asset, Errors.ASSET_NOT_LISTED
        );
    _reserves[asset].interestRateStrategyAddress = rateStrategyAddress;
  }
```

Listing 3.1: `Pool::setReserveInterestRateStrategyAddress()`

Recommendation Revise the above logic to apply the give `interestRateStrategyAddress` for the give reserve.

Status The issue has been fixed by the following commit: `360fd57`.

3.2 Improved Asset Addition Logic in ConfiguratorLogic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `ConfiguratorLogic`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

The `Extra Finance` protocol has a core `ConfiguratorLogic` contract to allow for dynamic update of protocol parameters. In the process of examining the related setters, we notice one specific setter can be improved.

In particular, we show below the implementation of this related setter routine, i.e., `executeInitReserve`. As the name indicates, it is used to initialize a reserve by creating and initializing `aToken`, `stable debt token`, and `variable debt token`. It comes to our attention that it requires the underlying token's decimals to be larger than 5 (line 56), which can be improved to additionally ensure the decimals are larger than `ReserveConfiguration.DEBT_CEILING_DECIMALS`.

```
30  function executeInitReserve(
31    IPool pool,
32    ConfiguratorInputTypes.InitReserveInput calldata input
33  ) public {
34    require(IERC20Detailed(input.underlyingAsset).decimals() > 5, Errors.
        INVALID_DECIMALS);
35
36    address aTokenProxyAddress = _initTokenWithProxy(
37      input.aTokenImpl,
38      abi.encodeWithSelector(
39        IInitializableAToken.initialize.selector,
40        pool,
41        input.treasury,
42        input.underlyingAsset,
      input.incentivesController,
      input.underlyingAssetDecimals,
      input.aTokenName,
      input.aTokenSymbol,
      input.params
    )
  );
  ...
}
```

Listing 3.2: `ConfiguratorLogic::executeInitReserve()`

Recommendation Revise the above setter to ensure the given reserve's underlying asset meets the minimal decimals requirement.

Status The issue has been fixed by the following commit: `360fd57`.

3.3 Incorrect Reward Calculation in ATokenRewardsReDistributionManager

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ATokenRewardsReDistributionManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The `Extra Finance` protocol brings in new staking component and enhanced debt payment. The new staking component is incentivized with extra reward distribution. While examining the associated incentive mechanism, we notice current approach to calculate reward amount should be revisited.

To elaborate, we show below the implementation of the related `getUnderlyigRewards()` routine. As the name indicates, this routine is used to query underlying rewards. It comes to our attention that the internal variable of `accumulatedRewardsIndex` is properly scaled by `PRECISION` (line 58). However, the calculated user reward amount is not properly scaled back (line 61). Note another routine `_claimUnderlyingRewards()` shares the same issue.

```
40  function getUnderlyigRewards(
41    address user
42  ) external view returns (address[] memory, uint256[] memory) {
43    (
44      address[] memory rewardsList,
45      uint256[] memory underlyingUnclaimedAmounts
```

```
    ) = _getAllPendingRewards();

    require(rewardsList.length == underlyingUnclaimedAmounts.length);

    uint256[] memory unclaimedAmounts = new uint256[](rewardsList.length);

    uint256 total = totalShares();
    uint256 share = userShares(user);

    for (uint i = 0; i < rewardsList.length; ++i) {
      if (total > 0 && underlyingUnclaimedAmounts[i] > 0) {
        uint256 accumulatedRewardsIndex = rewardsData[rewardsList[i]].
            accumulatedRewardsIndex +
          (PRECISION * underlyingUnclaimedAmounts[i]) /
          total;

        unclaimedAmounts[i] = (accumulatedRewardsIndex - userRewardsIndex[user]) * share
            ;
      }
    }

    return (rewardsList, unclaimedAmounts);
  }
```

Listing 3.3: `ATokenRewardsReDistributionManager::getUnderlyigRewards()`

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by the following commit: `81ffa34`.

3.4 Lack of MAX_ACCOUNTS_PER_USER Enforcement in ExtraXAccountFactory

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ExtraXAccountFactory`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Extra Finance` protocol is no exception. Specifically, if we examine the `ExtraXAccountFactory` contract, it has defined a specific risk parameter, i.e. `MAX_ACCOUNTS_PER_USER`. It is hardcoded to be 64 with the intention of ensuring each user will not create more than 64 accounts.

```
function createAccountFor(uint8 accType, address owner) public returns (address
    account) {
  uint256 id = nextAccountId[owner];

  // bytes memory dataToCall = abi.encodeWithSignature("createAccount(uint8,address,
      uint256)", accType, owner, id);
  bytes memory dataToCall = abi.encodeWithSelector(
    IAccountCreator.createAccount.selector,
    accType,
    owner,
    id
  );

  bytes memory result = Address.functionDelegateCall(LibAccountCreator, dataToCall);

  account = abi.decode(result, (address));

  accountsOfOwner[owner][id] = ExtraAccount(account, accType, id);
  accountInfo[account] = accountsOfOwner[owner][id];

  nextAccountId[owner] = id + 1;

  emit AccountCreated(owner, account, accType, id);
}
```

Listing 3.4: ExtraXAccountFactory::createAccountFor()

However, our analysis shows this risk parameter is not properly enforced when a new account is created. To elaborate, we show above the implementation of the affected `createAccountFor()` routine. This routine is used to create a new account for the given user. While it properly implements the intended logic, it can be improved by honoring the `MAX_ACCOUNTS_PER_USER` parameter by enforcing the following requirement, i.e., `require( id < MAX_ACCOUNTS_PER_USER)`.

Recommendation Enforce the `MAX_ACCOUNTS_PER_USER` parameter when a user account is created.

Status The issue has been fixed by the following commit: `360fd57`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Extra Finance` protocol, there are a few privileged `admin` accounts that play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
function setUnbackedMintCap(
  address asset,
  uint256 newUnbackedMintCap
) external override onlyRiskOrPoolAdmins {
  DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(
      asset);
  uint256 oldUnbackedMintCap = currentConfig.getUnbackedMintCap();
  currentConfig.setUnbackedMintCap(newUnbackedMintCap);
  _pool.setConfiguration(asset, currentConfig);
  emit UnbackedMintCapChanged(asset, oldUnbackedMintCap, newUnbackedMintCap);
}

/// @inheritdoc IPoolConfigurator
function setReserveInterestRateStrategyAddress(
  address asset,
  address newRateStrategyAddress
) external override onlyRiskOrPoolAdmins {
  DataTypes.ReserveData memory reserve = _pool.getReserveData(asset);
  address oldRateStrategyAddress = reserve.interestRateStrategyAddress;
  _pool.setReserveInterestRateStrategyAddress(asset, newRateStrategyAddress);
  emit ReserveInterestRateStrategyChanged(asset, oldRateStrategyAddress,
      newRateStrategyAddress);
}

/// @inheritdoc IPoolConfigurator
function setPoolPause(bool paused) external override onlyEmergencyAdmin {
  address[] memory reserves = _pool.getReservesList();

  for (uint256 i = 0; i < reserves.length; i++) {
    if (reserves[i] != address(0)) {
      setReservePause(reserves[i], paused);
    }
  }
}

/// @inheritdoc IPoolConfigurator
function updateBridgeProtocolFee(uint256 newBridgeProtocolFee) external override
    onlyPoolAdmin {
  require(
    newBridgeProtocolFee <= PercentageMath.PERCENTAGE_FACTOR,
    Errors.BRIDGE_PROTOCOL_FEE_INVALID
  );
  uint256 oldBridgeProtocolFee = _pool.BRIDGE_PROTOCOL_FEE();
  _pool.updateBridgeProtocolFee(newBridgeProtocolFee);
  emit BridgeProtocolFeeUpdated(oldBridgeProtocolFee, newBridgeProtocolFee);
}
```

Listing 3.5: Example Privileged Functions in the `PoolConfigurator` Contract

If these privileged `admin` accounts are managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Extra Finance` protocol, which forks from `AaveV3` – the popular decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. `Extra Finance` maintains the same core business logic, but reconstructs with new staking component and enhanced debt payment. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.