KING Protocol

Retail

SMART CONTRACT AUDIT

ECLIPTIC SECURITY

July 15th, 2025 | v. 1.0

PASS

Ecliptic's Security Team has concluded that this smart contract passes the security qualifications.

TABLE OF CONTENTS

Executive Summary . . . 3

Security Rating Calculation . . . 5

Technical Summary . . . 6

Scope . . . 7

Protocol overview . . . 8

Complete Analysis . . . 13

Test coverage and test results for all files . . . 23

EXECUTIVE SUMMARY

During the security audit, we examined the security of the smart contracts for the Retail module by the King Protocol team. Our task was to identify and describe any security issues within the platform's smart contracts. This report presents the findings of the security audit of the smart contracts with conducted between **June 4th** and **July 11th**.

Ecliptic team has conducted an audit for the smart contracts of the Retail module for the King Protocol team. The scope included the codebase of the module designed for a retail deposits into the King protocol without lower limit on a deposited amount. During the audit, auditors had focus on the funds flow, correctness of the asset allowlisting process, the correctness of integration with King protocol and correctness of price conversions and other related areas.

The security team raised several low-impact issues and sub-standard behaviors. However, all issues are resolved or verified by the King team. More details in the Complete Analysis section.

Also. the security team included several notes to be available in the report:

1) RetailCore is designed to properly work with King protocol smart contract.
The RetailCore is built on top of the King protocol and utilized the KING smart contract. All the calculations are performed on the King protocol's side.

However it also creates a backpropagating dependency: the Retail contract highly depends on a correctness of the King protocol itself - on its correct list of supported tokens, whitelisting functionality, King token minting during the deposit and correct balances calculation during the redemption.

2) Users might avoid unwrapping through the RetailCore.
The main purpose of the function unwrap() is to take a fee on each redemption of King tokens. However, the King implements a function redeem which can be used by anyone. Thus, the users can avoid using the function unwrap() and redeem the tokens using the function redeem() on the King protocol directly. The risk is known by the protocol's team: in the future, once they decide to turn on fees on, they will restrict the redeem function on the main contract to only whitelisted addresses.

3) High centralization level
The owner of the protocol is responsible for every core action in the contract: changing fees, setting limits on tokens, listing tokens (under the set of King-whitelisted tokens), changing epoch duration and resetting token limits. There are no limitations on the owner's actions - thus they can be performed any number of times in any intervals.

4) Single point of failure risk
The owner of the contract is a key role responsible for all core settings and operations. Thus, in case of owner's private key compromise, the whole protocol will be compromised with no failsafe mechanism. Therefore the security team highlights this design choice and highly recommends implementing the safe key storage for owner's private keys and usage of the multisig infrastructure to minimize the risk of compromise. The protocol's team ensured, that the admin will be set to the King protocol multisig

5) Upgradeable contracts
The Retail contract is upgradeable, thus the owner of the contract can change the logic at any moment. Thus, while the upgradeability is a regular practice adopted in web3 space, it still creates a controllable backdoor. Thus, the protocol should pay high attention to a safe storage of the proxy owner private key.

6) Extra gas spendings on certain users
By the design of the protocol, certain users will spend more gas on the deposit operation. It will happen in the end of the epoch, as limits clearing is bound to the deposit operation (based on an appropriate check). This operation has no direct security impact, however it will cause extra spending for some users, thus should be noted in the report.

7) Dependency on King's epochs
As it was mentioned, the protocol is dependent on the main King contract, and another instance of such dependence is the King's epochs update rate. The King Protocol has a limit on how many shares should be minted within the epoch. So, in case if Retail epochs are not synchronized with the King epochs, the shares' deficit may appear. Thus, epochs synchronization should be a point of monitoring for the Retail contract team.

SECURITY RATING CALCULATION

Approximate weight of unresolved issues.
Critical: -3 points
High: -2 points
Medium: -0.5 points
Low: -0.1 points
Informational: -0.1 point (in general, depends on the context)

Note: additional concerns, violated checklist items (including standard vulnerabilities), and verified backdoors may influence the final mark and weight of certain issues.

Starting with a perfect score of 10:

Critical issues: 0 issues (0 resolved): 0 points deducted

High issues: 0 issues (0 resolved): 0 points deducted

Medium issues: 0 issues (0 resolved): 0 points deducted

Low issues: 3 issues (3 resolved): 0 points deducted

Informational issues: 8 issues (5 resolved, 3 verified): 0 points deducted

Other:
- 0.1 points deducted for the centralization risk and single point of failure
- 0.1 points deducted for the controllable backdoor in a form of upgradeability

Security rating = 10 - 0.1 - 0.1 = 9.8

TECHNICAL SUMMARY

Contract Status

Issues Detected

	FOUND	FIXED/VERIFIED
Critical	0	0
High	0	0
Medium	0	0
Low	3	3
Info	8	8
TOTAL:	**11**	**11**

Best practices and optimizations

- 12 items resolved
- 0 items partially resolved
- 0 items unaddressed

Section is marked as **resolved**

SCOPE

Language/Technology: **Solidity**

Blockchain: **Ethereum**

The scope of the project includes:

- RetailCore.sol

Repository: **https://github.com/King-Protocol/king-minting-sc**
branch: **main**

Initial commit:

- 0bcc24b8ef108aeadfbb076f16b76cf62699bbdb

Final commit:

- 83da3c82e1603eaea167198431e291fdec3da8d0

Fixes merged into **main** branch:

- 3f221bef2a0dee4a8124041922426 54f126de08b

Last source code commits:

RetailCore.sol 294cd7926cb10ace81c17908c70a285391de5a05

PROTOCOL OVERVIEW

DESCRIPTION

RetailCore is a smart contract built on top of King Protocol. Contract implements core functions to deposit and redeems assets within the King Protocol. The functions allow users to deposit multiple collaterals (only whitelisted in the King protocol) into King and receive freshly minted KING tokens. The functions take fees on each deposit and redemption as well as implement an epoch-based limit per each token deposit. This limits restricts users from depositing tokens into a certain epoch if token's deposit cap is reached within this epoch.

Core functions:

1. deposit()

Input:

- tokens: list of tokens to deposit into the King protocol. Must be whitelisted within the King.
- amounts: amount of each token to deposit.

Prerequisites: User has approved all the tokens to the RetailCore Smart contract.
Description: Function `deposit` allows users to deposit specified tokens in the King protocol and receive King tokens. Each token has a certain limit per epoch. Each epoch is divided into an equal period of time. The function takes a fee in the KING token, specified by the `depositFeeBps` variables.
Note: Amount of KING tokens to be minted is calculated on the "King protocol" smart contract.

2. unwrap()

Input:

- kingAmount: amount of KING tokens to redeem.

Prerequisites: User has approved KING tokens to the RetailCore Smart contract.
Description: Function `unwrap` allows users to redeem their KING tokens and receive multiple underlying assets. All underlying tokens are sent to the msg.sender's address. The function takes a fee in the KING token, specified by the `unwrapFeeBps` variables.

Epoch-based deposit approach: Users can deposit tokens within the current active epoch. Epoch affects the amount of tokens which can be deposited. Each token has a certain limit. Users are able to deposit a token until it reaches its limit. After that, users will have to wait until another epoch has started. Epochs are changed automatically during deposit. Before the core logic of deposit is executed, it is validated if a new epoch has started. The contract updates the storage variables connected to the epoch logic such as:

- Reset limit per each token;
- Update next epoch timestamp.

ROLES AND RESPONSIBILITIES

RetailCore.sol

1. Default Admin

DEFAULT_ADMIN_ROLE from OZ AccessControl. Default admin is set during the initialization of smart contract to a parameter `_admin`.
In the deployment script "./scripts/deployment/deploy_retailcore.ts" "_admin" is set to the KING protocol mulisig 0xF46D3734564ef9a5a16fC3B1216831a28f78e2B5..
Responsibilities:

- Manages other roles of the RetailCore;
- Sets the settings of the RetailCore including epoch duration, deposit limit per token, pause tokens for deposit, set deposit and withdraw fee;
- Withdraws collected fees in KING tokens;
- Resets the epoch forcibly and update the price provider.

Note: The contract can have multiple Default Admins.

2. Pauser

Standard implementation compatible with OZ AccessControl. It is able to pause/unpause the deposit() function.
By default, the role is set During the initialization to the parameter '_admin'.
Note: The contract can have multiple Pausers.

3. User

Users are able to execute function deposit() and unwrap(). Users should have enough tokens on their balances and grant a sufficient allowance to the RetailCore before calling any function.

LIST OF VALUABLE ASSETS

1. KING token

- the protocol transfers the KING token from users to the King protocol (redeem operation) and King protocol mints fresh KING tokens to the users (deposit operation)
- the protocol keeps fee accumulated in King token, and the owner can withdraw that fee at any time

Note: however, any direct KING transfer to the contract will be locked, as the withdraw functionality relies on the tracked amount stored in the storage variable.

2. Underlying tokens

- list of underlying tokens is regulated by the King protocol, and the Retail relies on that list
- the contract transfers all tokens from users to the King protocol or vice versa with no expectation of balance keeping on the contract

Note: however, the contract relies on the King protocol in both operations:

- during the deposit the contract relies on the fact that King will take the exact amount of each approved token (with no dust left)
- during the redemption the contract relies on the fact that King will transfer the exact amounts as it reported to the Retail

Note: The contract implements no rescue functionality, thus any direct transfer to the contract or any dust left after operations will be locked.

SETTINGS

OZ pause: standard OZ pause inherited via the Pausable contract. Is available for the Pauser role and responsible for pause/unpause of deposits. By default deposits are unpaused.

kingContract: Address of the “King Protocol” SC on top of which the RetailCore is built. It is set only during the initialization.
Default value: Deploy scripts set it to 0x8F08B70456eb22f6109F57b8fafE862ED28E6040

depositFeeBps: Fee percentage taken on each deposit. 1 BPS equals 0.01%. It is set during initialization and can be set by the Default Admin at any time.
Default value: Deployment scripts set it to 0%

unwrapFeeBps: Fee percentage taken on each unwrapping. 1 BPS equals 0.01%. It is set during initialization and can be set by the Default Admin at any time.
Default value: Deployment scripts set it to 0%

epochDuration: A time duration which indicates how each epoch lasts. It is set during initialization and can be set by the Default Admin at any time.
Default value: Deployment scripts set it to 1 week

depositLimit: A maximum possible amount per each token which is possible to use in the epoch. 0 means that the tokens can’t be used. Limits can be set by the Default Admin.
Default value: 0 for every token (all tokens are disabled)

tokenPaused: A flag which indicates if a certain token is paused for the depositing. It is set by the Default Admin.
Default value: all tokens are unpaused**defaultFeeBps**: default value of fee bps, used when

DEPLOYMENT
Deployment script for Ethereum is located at:
“./scripts/deployment/deploy_retailcore.ts”.

The script deploys the RetailCore as a transparent upgradable proxy and sets deposit limits for tokens.

STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Issues tagged “Verified” contain unclear or suspicious functionality that either needs explanation from the Customer’s side or it is an issue that the Customer disregards as an issue. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

LOW-1 | RESOLVED

Absence of revert on zero amount

RetailCore.sol: depositMultiple()

The cycle contains the line: “if (amountToDeposit == 0) continue;”

Thus the cycle will skip the array intros with token amount 0. However, the revert is more preferable in such situations:

- there are no requirements on tokens order (on both Retail and King contracts);
- deposit is later redirected to the King protocol where there are no restrictions on specific tokens order as well;
- every 0 amount entry increases gas spending with no pros for the user
- 0 amount creates a place for data entry error (either human mistake or incorrect interface)

The issue is marked as Low, as while there is no direct security impact, it still creates a space for an error and ambiguity (especially on the edge of integration with another protocol)

Recommendation: Revert on 0 amount

LOW-2 | RESOLVED

Extra argument during the initialization

RetailCore.sol: initialize(), _admin

- the initialization function triggers updatePriceProvider() and 3 setters available to the admin only;
- thus it requires to have the admin set in advance
- initialization is triggered during the deployment, thus the only possible initial admin is a deployer
- deployer is passed as _admin during the initialization

Thus, the parameter is unnecessary, as the initial admin can be set to msg.sender and that is the only available option based on the contract structure.

Recommendation: Remove unnecessary parameter as it has only 1 possible option to be set

Post-audit: The workaround was added to set the deployer as an admin and revoke the role after the initializer operations. The deployment script now sets the King multisig (0xF46D3734564ef9a5a16fC3B1216831a28f78e2B5) as “_admin” parameter.

LOW-3 | RESOLVED

No validation on the epoch duration

RetailCore.sol: setEpochDuration()

Currently the function does not validate the duration, thus it can be set for several seconds or several years. Thus it creates a possibility of misconfiguration.

The issue is marked as Low, as while there is an impact on user flow, the setter is controlled by owner, narrowing the misconfiguration risk to a human error

Recommendation: Consider adding min/max limits for the duration.

Post-audit: Epoch is validated to be between 1 hour and 30 days.

INFO-1 | RESOLVED

Maximum allowed fee can be set to 90%

RetailCore.sol: setDepositFeeBps(), setUnwrapFeeBps().

Setters allow to specify the fee percentage taken upon each deposit or unwrapping. The maximum allowed fee is currently 90%, which might allow the Default Admin to set large fee, which would take almost all the tokens.

Recommendation: Verify the allowed fee limit and consider lowering the maximum allowed fee to a realistic value suitable for the protocol. Ensure that the protocol's UI accurately reflects the possible maximum value of the fee.

Post-audit: The maximum allowed fee was decreased to 50%

INFO-2 | RESOLVED

Comment suggesting that logic is not finished
RetailCore.sol: _getCurrentEffectiveNextEpochTimestamp(), line 391.

The comment `Or potentially return storedNextTimestamp if that's desired behavior for "infinite"` suggests that the logic is not finalized.

Recommendation: Validate if current logic is correct or update it. Remove the comment.

Post-audit: The section was removed, as the check against 0 for the epochDuration is already enforced in the code.

INFO-3 | RESOLVED

Repeated tokens can be passed in the array
depositMultiple()
Function depositMultiple() accepts the array of tokens without validating that the array consists of unique addresses. Though, this doesn't affect the security of the contract, having repeated addresses in the array will revert due to insufficient allowance. This happens when tokens are approved to the King Contract (line 166). Since the allowance is re-written each time, the final allowance of the repeated token will be equal to the last amount corresponding to the token. This might produce unclear error which wouldn't suggest that the issue is caused due to repeated tokens in the array.
Note: the same issue applies to the setLimits() function as well

Recommendation: Validate that tokens do not repeat in the array..

INFO-4 | VERIFIED

Contract relies on change of balance on the King protocol instead of own balances
unwrap()

The function expects, that the King protocol will return certain amounts of underlying tokens, thus the function checks the balance of each toke before and after the redeem from the King protocol. However, the function relies on the balances change reported by the King protocol, not on the balances change on the Retail contract.

While the contract is a superstructure around the King protocol and thus trusts into the return values, there is no guarantee that the change of balances on the King protocol during the redemption operation will correspond to the received amounts of tokens.

The issue is marked as Info, as while there is a certain level of risk associated with relying on the balance change reported from an integrated protocol, the retail contract is built around that protocol and thus relies on trust.

Recommendation: Consider implementation of balance change check directly on the Retail contract instead of relying on the reported balance check from the King protocol.

Post-audit: the King team verified that the approach is intended and the risk connected with the King contract balance dependency is acceptable, as no significant changes to the King contract's logic will be introduced without changes in the Retail contract.

INFO-5 | RESOLVED

Owner can reset token limits even before the end of the epoch
RetailCore.sol: resetEpoch()

The contract implements limits on token deposits, and those limits are renewed at the start of every epoch and are active up to the end of each epoch. The reset of the epoch limits is processed as additional action during each deposit - in case if it is necessary. However, the owner of the contract can reset epoch limits any time, even if epoch is not finished yet.

The issue is marked as Info, as it refers to the substandard business logic and responsibilities of the admin of the contract, thus requires clarification from the team

Recommendation: Verify if owner should have the ability to clear token limits before the epoch end.

Post-audit: functionality was resolved. However, it should be noted that the admin still has this ability via the epoch duration setting, as it allows resetting the epoch within the same call.

INFO-6 | RESOLVED

Unsafe price provider can be used for token prices
RetailCore.sol: forceUpdatePriceProvider()
The contract relies on the price provider inherited from the King protocol. However, the contract allows 3rd-party price provider to be set, thus breaking the integration with King protocol.

The issue is marked as Info, it refers to the substandard business logic and responsibilities of the admin of the contract, thus requires clarification from the team. And also, while there is no direct usage of the price for onchain operations, it is expected to be used by the dApp, thus it may create wrong price calculation for the users (wrong - not the same as within King protocol)

Recommendation: Verify if the 3rd party price feeds should be allowed

Post-audit: method was removed from the contract, and only the native King price provider can be used now

Extra price conversion functionality
RetailCore.sol: priceProvider, _tokenAmountToUsd(), _ethToUsd(), tokenAmountToUsd()

Looks like the contract contains a large part of the logic connected to the price feed and price conversion for the used tokens. However, price conversion logic is not used for the protocol operations - neither for the deposit nor for the redemption (as all price calculations are performed within the King protocol). Looks like the only purpose of price provider storage and price conversion logic is for the usage within the dApp via an additional view method.

However, such practice is substandard, as usually either direct price extraction is used (from the King protocol in the case of Retail contract), or the dApp integrated appropriate offchain connectors to price feeds based on the token lists. Thus looks like the contract creates the unnecessary complexity for the price extraction with minimal utilization

Recommendation: Verify that price provider and price conversion functionality is necessary, is utilized by the dApp, and does not substitute the direct price extraction from the King protocol.

Post-audit: The King team verified that the price logic implementation is intended to be used on the dApp side to simplify the frontend part of the protocol.

INFO-8 | RESOLVED

Dust amount check may be considered
RetailCore.sol: depositMultiple(), unwrap()

Both core operations revert in case of 0 net KING amount (after the fee deduction). However, looks like a certain dust amount check would be more suitable for the protocol. Currently the user may proceed with the deposit of couple of wei of the underlying tokens token (to receive weis of the King token), or redemption of the couple of weis of the King token. Thus the gas fee will exceed the received profit, not saying about the loss of accuracy due to rounding errors.

Recommendation: Consider implementation of a certain dust limit on the King amount to be redeemed or received during the deposit

Post-audit: the protocol's team verified that the exact check as enough as the goal of the protocol is to work with "dust" amounts.

1. Use the “delete” keyword.
RetailCore.sol: _resetEpoch(), line 374.
Usage of “delete” is a proper way of resetting the value inside mappings instead of assigning zero values.

2. Pass local variable in the event.
RetailCore.sol: _resetEpoch(), line 376.
Variable `newNextTimestamp` can be passed in the event. This reduces gas costs due to reading from storage.

3. Use created local variable instead of value from array.
RetailCore.sol: depositMultiple(), lines 165, 166.
Created variable `tokenAddress` can be used instead of `tokens[i]`. This increases the readability of the code and potentially reduces the gas spendings.

4. Variable can be assigned a value during creation.
RetailCore.sol: unwrap(), line 197, 199.
Variable `netKingAmount` is created without assigning a value. The value is assigned to it in the next line. Assigned the value during the creation of the variable in order to reduce the lines of code and increase readability.

5. Duplicate of the whitelist check
depositMultiple() contains kingContract.isTokenWhitelisted() check which duplicates the same check in the kingContract._deposit(). While the existence of the check is justified (as the one cannot fully rely on a 3rd party protocol), it still creates extra gas spending.

6. Unnecessary check
unwrap(): if (allTokens.length != amountsBefore.length || allTokens.length != amountsAfter.length)
The contract relies on the correctness of King protocol, thus this check may be considered as extra, since the function returns same length array every time and no token is removed from the array during the redeem() operation.

7. depositMultiple(): no limit on the array length.
Combined with the absence of the repeatable token check that may create the unnecessary gas spending. Consider limiting the length of the array.

8. Extra check
depositMultiple(): “if (allowedLimit == 0 ...”
Since the function already checks that amountDeposit cannot be 0, check for 0 limit is already covered by next check in the if.

9. isTokenPaused() duplicates the tokenPaused autogenerated viewer

10. Absent address(0) check for setDepositLimits()

11. Extra condition
_updateEpochIfNeeded(), _getCurrentEffectiveNextEpochTimestamp() contain checks for epochDuration to be greater than 0. However this fact is guaranteed by setEpochDuration() and the initial duration set during the contract construction. Thus the check can be safely omitted.

12. Good practice is to have constructor() with initializers disabled to avoid any storage confusion on implementations.

Post-audit:
Resolved items: all items resolved
Partially resolved items: -
Unaddressed items: -

Thus the section is marked as **resolved**

Storage structure and data modification flow	Pass
Access control structure, roles existing in the system	Pass
Public interface and restrictions based on the roles system	Pass
General Denial Of Service (DOS)	Pass
Entropy Illusion (Lack of Randomness)	N/A
Order-dependency and time-dependency of operations	Pass
Accuracy loss, incorrect math/formulas other violated operations with numbers	Pass
Validation of function parameters, inputs validation	Pass
Asset management, funds flow and assets conversions	Pass
Signatures replay and multisig schemes security	N/A
Asset Security (backdoors connected to underlying assets)	**Failed**
Incorrect minting, initial supply or other conditions for assets issuance	Pass
Global settings mis-using, incorrect default values	Pass
3rd party dependencies, used libraries and packages structure	Pass
Single point of failure	**Failed**
Centralization risk	**Failed**
Violated communication between components/modules, broken co-dependencies	Pass
General code structure checks and correspondence to best practices	Pass
Language-specific checks	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by the King team

As a part of our work in verifying the correctness of the protocol's code, our team has checked the complete set of unit tests prepared by the protocol's team.

The protocol's team has prepared a solid coverage covering key user flows in the system. All of them were also carefully checked by the team of auditors.

Tests written by the Ecliptic team

During the audit the security team performed an extensive testing of the system, with main focus on the correct integration with King protocol:

- set of tests for the unbalanced deposit with checking of the influence on King protocol pools
- set of fuzzy tests on the King pool balance as a result of unbalanced deposit via the Retail module
- set of tests on a correct epoch update and synchronization of epochs between the Retail and the King
- general set of tests with mai user flows
- edge cases and hypotheses testing
- tests on the access control structure

We are grateful to have been given the opportunity to work with the audited protocol team.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

Ecliptic's Security Team recommends that the protocol’s team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

ECLIPTIC SECURITY