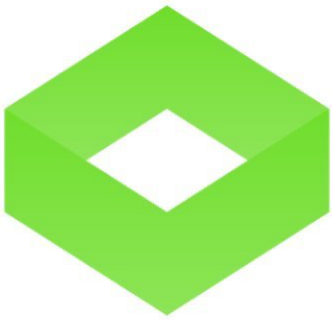
verichains

SECURITY AUDIT OF

KROMA SPECTRUM CONTRACTS



Public Report

May 17, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
BSC	Binance Smart Chain or BSC is an innovative solution for introducing interoperability and programmability on Binance Chain.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Polygon	Polygon is a protocol and a framework for building and connecting Ethereum-compatible blockchain networks. Aggregating scalable solutions on Ethereum supporting a multi-chain Ethereum ecosystem.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on May 17, 2024. We would like to thank the Kroma for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Kroma Spectrum Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Kroma Spectrum Contracts

Spectrum is Kroma's native staking service designed to enable high yields through restaking while utilizing staking tokens in a diverse array of games and DeFi ecosystems on Kroma.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Kroma Spectrum Contracts. It was conducted on commit `6e21c92216006ecfa19dea228d26aa579fc51016` from git repository link: *https://github.com/kroma-network/eco-contracts*

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
573844f1c83100abb851bbe464883ee5126daf12a877b6f1129bb20498a91d2a	core/SpectrumCore.sol
d0b02e578eca70de8ec50f81766a5e92a16f1638d75c425ef8a857f514c410a3	core/SpectrumCoreEntry.sol
51388a6af2f4c98301e549b01c363a9d73d1ecfc07630bdde12af45d6d06d6f4	core/SpectrumCoreLogic.sol
193cef59b3d687e66916b6448c12f86149d32354a669a8f1aef9b0a4c16237ac	core/SpectrumCoreState.sol
7290aae964d86c4c4fb94a2182a2bd1d4293f634cfb5ccb077b5a0a560b994d3	core/SpectrumCoreUtils.sol
79d4565c9d2fa5fbcc7c4aae6657f6037610f69bc3af210ece5b16c2ca1c987c	hub/SpectrumHub.sol
5e9b8cf987dab624a8d7f856108da479069e75c0a99394b041627b60bf116251	hub/SpectrumHubEntry.sol
d6c5c32538fac822dd39f2d103e8e43fb04f60abedf3dec8978bfce12b211da5	hub/SpectrumHubLogic.sol
069be5efc675a88b58ba63b45da500421a039736b26cd3d14f9b823440785b5f	hub/SpectrumHubState.sol
6cb7f98b0264327a385de75fef79097481fed633498adb9a7b1b9e13b3425be4	hub/SpectrumHubUtils.sol
c19570c3c0456cd76b7182353ae5d02a7ed4f8a5a46649cdeedf3b882550fe1d	SpectrumETH.sol
ef5d199fb73607619ed1eaf0f9df21541c7a499b7dfb9a92b18a5d0ba566c4d1	common.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Kroma acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Kroma understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Kroma agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Kroma will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Kroma, the final report will be considered fully accepted by the Kroma without the signature.

2. AUDIT RESULT

2.1. Overview

The Kroma Spectrum Contracts was developed using the `Solidity` language, with a required version of `^0.8.22`.

2.1.1. Spectrum Hub Contract

The hub contract facilitates staking of supported tokens for Spectrum ERC4626 tokens. It routes supported tokens through vendor contracts (e.g., EtherFi, Lido, Uniswap) to acquire WeETH tokens. The WeETH tokens are subsequently exchanged for Spectrum tokens within the Spectrum Core contract.

The hub contract supports a list of stakeable and unstakeable tokens, managed via the manage functions within the SpectrumHub state abstract contract. Unstaking currently permits users to convert Spectrum tokens into WeETH.

2.1.2. Spectrum Core Contract

The contract governs the Spectrum token, adhering to the ERC-4626 standard. This token is minted proportionate to staked tokens and assets within the Core Contract. Although designed for flexibility with various assets, the contract currently exclusively interfaces with WeETH tokens. The contract handles the final step of the Spectrum stake and unstake process within the Spectrum Hub contract. Only the hub contract can initiate the stake and send Spectrum tokens to the user.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Kroma Spectrum Contracts.

Issue	Severity	Status
SpectrumCore - Inflation attack	HIGH	Acknowledged
SpectrumCore - Stake logic break when admin add more support tokens	MEDIUM	Acknowledged
SpectrumCore - Feature not working in declare Solidity version	LOW	Fixed

Issue	Severity	Status
SpectrumCoreState - Not verify asset when updating	LOW	Fixed
SpectrumHub, SpectrumCore - Mistake logic inside contructor function	LOW	New
SpectrumCoreLogic - Unreachable code in `_evalStakeRequest` function	INFORMATIVE	Acknowledged
Unuse UbpMath library	INFORMATIVE	Fixed

Table 2. The issue list

2.2.1. SpectrumCore - Inflation attack HIGH

The first stake in SpectrumCore will mint an amount of SP equivalent to the amount of the Asset token. If an attacker stakes 1 Asset, they will receive 1 SP, corresponding to a 100% share of the Vault. For any subsequent stakes with an amount `x` of the Asset token, the attacker can front-run by transferring `x` Asset tokens to this contract. This causes the contract to round down the SP amount the user should receive to zero. Consequently, the attacker, holding 1 SP, can claim all assets in the vault, including both the attacker's and the user's assets just sent.

```
function _getSpectrumFromAsset(
        SnapGlobal memory gSnap,
        address asset,
        uint256 assetAmount
    ) internal view returns (uint256 spAmount) {
        CoreSupportAssetInfo memory assetInfo =
_getSpectrumCoreStorage().assetInfos[asset];
        uint256 spTotalSupply = gSnap.info.spETH.totalSupply();
        if (spTotalSupply == 0) return assetAmount; //@VerichainsAudit: First deposit will
mint amount of SP equals the amount of Asset token
        spAmount =
            (spTotalSupply * _switchAssetEval(assetInfo.id)(assetInfo.assetEvaluator,
assetAmount)) /
            gSnap.spSnap.evalValue;
        //@VerichainsAudit: When spTotalSupply is 1, attacker can front-run to transfer x
asset amount to this contract and cause the contract will round down the sp amount user
should receive to zero
    }
```

RECOMMENDATION

The deployer should be the first to stake in SpectrumCore to prevent an inflation attack.

UPDATES

- **May 17, 2024**: This issue has been acknowledged by Kroma team.

2.2.2. SpectrumCore - Stake logic break when admin add more support tokens MEDIUM

The `addCoreSupportAsset` function allows an admin to add additional support tokens to the SpectrumCore. However, within the staking flow, the contract only supports `WEETH` and will revert when calculating any other token in the SupportAssetInfo. This issue disrupts the staking logic and causes the contract to revert. Therefore, if an admin adds more support tokens to the SpectrumCore, the contract will not function as expected.

```
function addCoreSupportAsset(address asset, address assetEvaluator) external onlyAdmin {
        CoreSupportAssetInfo storage assetInfo =
_getSpectrumCoreStorage().assetInfos[asset];
        require(assetInfo.assetEvaluator == address(0), "SpectrumCore: asset already
exists");
        assetInfo.assetEvaluator = assetEvaluator;
        assetInfo.id = _getSpectrumCoreStorage().supportAssets.length;
        _getSpectrumCoreStorage().supportAssets.push(asset); //@VerichainsAudit: Add more
support token to the SpectrumCore
    }

    function getGlobalSnap() public view returns (SnapGlobal memory gSnap) {
        SpectrumCoreStorage storage $ = _getSpectrumCoreStorage();
        gSnap.info = $.info;
        gSnap.assets = $.supportAssets.values();
        gSnap.snaps = new SnapAsset[](gSnap.assets.length);

        unchecked {
            CoreSupportAssetInfo memory assetInfo;
            for (uint256 i; i < gSnap.snaps.length; i++) {
                assetInfo = $.assetInfos[gSnap.assets[i]];
                gSnap.snaps[i].evalValue = _switchAssetEval(assetInfo.id)(
//@VerichainsAudit: check all support token
                    assetInfo.assetEvaluator,
                    IEcoERC20(gSnap.assets[i]).balanceOf(address(this))
                );
                gSnap.spSnap.evalValue += gSnap.snaps[i].evalValue;
            }
        }
    }

    function _switchAssetEval(
        AssetID id
    ) internal pure returns (function(address, uint256) internal view returns (uint256)) {
        if (id == AssetID.WEETH) return _evalWEETH;
        revert Spectrum(SpectrumErrors.Function); //@VerichainsAudit: Revert with other
```

```
tokens
    }
```

UPDATES

- **May 17, 2024**: This issue has been acknowledged by Kroma team.

2.2.3. SpectrumCore - Feature not working in declare Solidity version LOW

The named parameter is not supported in Solidity below 0.8.18. So the contract will not work as expected in the declare Solidity range version below 0.8.18.

```
pragma solidity ^0.8.0;

    struct SpectrumCoreStorage {
        CoreInfo info;
        EnumerableSet.AddressSet supportAssets;
        mapping(address asset => CoreSupportAssetInfo) assetInfos; //@VerichainsAudit:
named parameter is not supported in Solidity below 0.8.18
    }
```

RECOMMENDATION

Update the Solidity version to the latest available release. Upgrading to the newest version will ensure compatibility with the latest features, improvements, and security enhancements.

UPDATES

- **May 17, 2024**: This issue has been acknowledged and fixed by Kroma team.

2.2.4. SpectrumCoreState - Not verify asset when updating LOW

The `updateCoreSupportAsset` function does not verify whether the asset is supported. As a result, an admin can update the SpectrumCore to include an unsupported asset. The team should update the logic to ensure this verification is clear and to prevent junk data from being included in the contract.

```
function updateCoreSupportAsset(address asset, CoreSupportAssetInfo memory assetInfo)
public onlyAdmin {
        SpectrumCoreStorage storage $ = _getSpectrumCoreStorage();
        //@VerichainsAudit: Not check asset is supported
        _checkAssetInfoNotEmpty(assetInfo);
        $.assetInfos[asset] = assetInfo;
    }
```

RECOMMENDATION

The team should update the logic to ensure that the asset is supported before updating the SpectrumCore. This will help prevent junk data from being included in the contract.

UPDATES

- **May 17, 2024**: This issue has been acknowledged and fixed by Kroma team.

2.2.5. SpectrumHub, SpectrumCore - Mistake logic inside contructor function LOW

In the project, SpectrumHub and SpectrumCore are implemented using a Proxy contract. As a result, the initialize function, typically called in the constructor, only affects the implementation contract. This can lead to the deployer mistakenly omitting the initialize call for the proxy itself.

To address this, the team should update the logic to guarantee proper execution of initialize for the proxy. Alternatively, within the implementation contract, consider using __disableInitializers() to prevent it from being invoked in place of the intended initialize function.

```
contract SpectrumHub is SpectrumHubEntry, MulticallUpgradeable {
        constructor() {
            initSelectorRoleControl(_msgSender());
        }
    }
```

2.2.6. SpectrumCoreLogic - Unreachable code in `_evalStakeRequest` function INFORMATIVE

The `_evalStakeRequest` function contains unreachable code. In the else-if branch, the function is designed to return the asset amount that a user should stake to receive a specific SP amount. However, the staking flow in SpectrumCoreEntry only allows users to calculate based on the asset amount instead.

```
function _evalStakeRequest(
    SnapGlobal memory gSnap,
    RequestParams memory reqParams
) internal view returns (RequestParams memory) {
    if (reqParams.from == address(0) || reqParams.to == address(0)) revert
Spectrum(SpectrumErrors.Address);

    if (reqParams.assetAmount == 0 && reqParams.spAmount == 0) revert
Spectrum(SpectrumErrors.Amount);
    else if (reqParams.assetAmount == 0) //@VerichainsAudit: this branch is unreachable
        reqParams.assetAmount = _getAssetFromSpectrum(gSnap, reqParams.asset,
reqParams.spAmount);
    else {
        reqParams.spAmount = _getSpectrumFromAsset(gSnap, reqParams.asset,
```


```
reqParams.assetAmount);
    }

    return reqParams;
}
```

To more accurately reflect the function's purpose, the team should remove the unreachable code or update the function's logic to ensure it functions as intended.

UPDATES

- **May 17, 2024**: This issue has been acknowledged by Kroma team.

2.2.7. Unuse UbpMath library INFORMATIVE

The UbpMath library is not utilized in the SpectrumCore and SpectrumHub contract. The team should consider removing the library to decrease the contract's size and enhance readability.

UPDATES

- **May 17, 2024**: This issue has been acknowledged and fixed by Kroma team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*May 17, 2024*	Public Report	Verichains Lab

Table 3. Report versions history