verichains

SECURITY AUDIT OF

WEMIX ON KROMA SMART CONTRACTS

WEMIX on KROMA

Public Report

Jan 12, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
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.
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.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jan 12, 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 WEMIX on Kroma smart 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 found some vulnerabilities in the given version of WEMIX on Kroma smart contracts. Kroma team has resolved and fixed some of these issues following our recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About WEMIX on Kroma smart contracts

WEMIX on Kroma is a cross-chain staking protocol for WEMIX Coin. WEMIX on Kroma is operated by Lightscale Pte. Ltd., a company which is building Kroma, the first OP stack rollup with active fault proofs using zkEVM.

WEMIX on Kroma enables the users of WEMIX3.0 to discover Ethereum's ecosystem, with the help of the Layer 2 technology powered by Kroma.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of WEMIX on Kroma smart contracts.

It was conducted on commit `bfecd3dc551d7aa5c95046b8c176e05bf8712e96` from git repository link: *https://github.com/light-scale/wemix-on-kroma*.

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

SHA256 Sum	File
04078a4e5ab06c52249df392d62bb2b9f3d672b566ed4f7c73b56f4517df5e32	./token/ERC20/EcoERC20.sol
9e5946844ca38049f45fee36aee73ee012b99645f05d171932abf25f8418ae42	./stake/deploy.sol
52ee4f527b1b3b23c383c197a73750cdf56766cf9f4826c8d136a454639f8f53	./stake/TokenizeStakeBase.sol
23e8a56fe1469c15469652b9f05f0fcb24c0e0d13a783544e5c5edf0ed878c08	./access/AdminableProxy.sol
049c03ac0f85f3356c1ecde48174036c51f15f4c40f83b65be22248570aa5504	./access/SlotAdminable.sol
584c5d922b6c382d2a091e99516ae1eba6597c96067542938365b67d4cdcf148	./access/SlotServiceSigner.sol
6e19ac57a8f0f0bcceb4faa215921ada323fcd3a0cb01164136cdfb0251030b3	./access/SlotPausable.sol
e5159c2a683541e1322a219ab886819bd25168f719f33ce53c896fa26489d84f	./cross-bridge/CrossBridgeBase.sol
d15a364c5df26cf77bba56f4f3b30bde4f7efd64f1ab573e5af422e98a783d2c	./cross-bridge/CrossBridgeRemote.sol

acd8f4ce4290d27ca93a631568991604a9cc9f90de43db987514e75d9dca8f46	./cross-bridge/CrossBridgeOrigin.sol
a56f209b661add29ef4754e347650a616e2e462d8b97230b6097e72e0fa64342	./cross-bridge/CrossDeploy.sol
52c1c1bf6306d27aae87861d664ca4a71a96878ef23f3dba38b61dcf3d87cb20	./wemix/wonder/interfaces/IRewarder.sol
8877efb3217c4be3be7b5b4878ef2fb1758fe6a48a8cf9270864958dd8c6cbb5	./wemix/wonder/interfaces/IWithdrawalNFT.sol
79fdafe43ff81358c5d76e35b82f50e54f88b9348fa2d2642ae7aaca44837f87	./wemix/wonder/interfaces/INCPStaking.sol
83093144937702c1ec57e09c7f2628bb74403ae7e48b3f1a2661fec30b9d34fe	./interfaces/IEcoERC20.sol
65b2a4be1396944ef1b73a24f58d9c8e606b7aaa8b3e05a7fc7d6b21e66128a5	./interfaces/UniversalTypes.sol
d728a435b87154a1eed45139c0dc4165e247a2f51a52ee29bd9008335a10c4d5	./interfaces/ISlotAdminable.sol
6d6493e4660b0c2fef57c0e7c8ef1d8823f9917ace039363d3774231258fed07	./interfaces/IEXBridge.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 WEMIX on Kroma smart contracts was written in `Solidity` language. The source code was written based on OpenZeppelin's library.

WEMIX on Kroma provides two primary services:

- **Cross-chain Transfer Service**: Facilitates the transfer of WEMIX coins between the WEMIX3.0 network and the Kroma network. Only WEMIX coin is supported.
- **Liquid Staking Service**: Offers liquid staking for WEMIX.e tokens on the Kroma network.

2.1.1. Cross-chain Transfer Service

Users can deposit their WEMIX coins from the WEMIX3.0 network to the Kroma network, receiving WEMIX.e tokens in return. This process typically completes in about 20 seconds.

When depositing WEMIX coins, a small portion of the transfer amount is converted to ETH (`amountForNativeAlloc`) to enhance user experience, as Ethereum serves as the native gas token of the Kroma network.

There is a small fee for depositing WEMIX coins from the WEMIX3.0 network to the Kroma network. The fee is the greater between 1 WEMIX and 0.1% of the total transfer amount.

Deposited WEMIX coins are automatically staked in WONDER Staking of WEMIX3.0, earning rewards. These rewards generate new WEMIX.e tokens on the Kroma network daily, and the newly minted WEMIX.e tokens flow into the Liquid Staking Vault for stWEMIX.e, thereby increasing its value.

Users can withdraw WEMIX.e tokens from Kroma network and receive WEMIX coins on the WEMIX3.0 network. The withdrawal process takes approximately 7 days, accounting for the unstaking time required in WONDER Staking on WEMIX3.0. Once initiated, the withdrawal process is irreversible.

Users must manually claim their withdrawn WEMIX coins on the WEMIX3.0 network after the 7-day waiting period.

Please notice that the security of the relay process and withdrawal duration verification are out of scope for this audit.

2.1.2. Liquid Staking Service

Unlike conventional staking services, liquid staking allows users to convert their deposited assets into equivalent ERC-20 tokens, providing more flexibility.

Users can exchange their WEMIX.e tokens to stWEMIX.e tokens on the Kroma network by using this liquid staking service. The exchange rate of stWEMIX.e to WEMIX.e continuously increases over time, reflecting the rewards from WONDER Staking. The rewards are used to mint additional WEMIX.e tokens on the Kroma network, proportionate to the earnings. These tokens are then utilized to enhance the value of stWEMIX.e based on a set ratio (initially, 100%).

In addition to the increase in the value of stWEMIX.e, users who exchange WEMIX.e for stWEMIX.e, as well as stWEMIX.e holders, will receive WEMIX Community Points, redeemable for KRO tokens in the future.

Please notice that these staking parameters (such as reward/fee ratio) are controlled by the contract admin and are out of scope for this audit.

2.2. Findings

During the audit process, the audit team found some minor vulnerabilities in the given version of WEMIX on Kroma smart contracts.

Kroma team fixed some issues, according to Verichains's draft report, in commit `5b8cd6944ca5fb685e49b600f67ee01671a2e7cb`.

#	Issue	Severity	Status
1	Missing `bridgeTotalSyncStaking` update when invoking `toOriginSyncExtra`	MEDIUM	Fixed
2	`setRemoteToken` will fail if the owner is changed	MEDIUM	Fixed
3	`setStakingInfo` will fail if the owner is changed	MEDIUM	Fixed
4	Duplicated usage of staking value if `ncpIds.length > 1`	MEDIUM	Fixed
5	Incomplete implementation for `SupportUnderlyingNative` contract	LOW	Fixed
6	Incomplete implementation for `SupportRewardNative` contract	LOW	Fixed

#	Issue	Severity	Status
7	Upgradeable contract but based on non-upgradeable ones	LOW	Acknowledged
8	Incorrect initialization implementation for `EcoERC20Pausable` contract	LOW	Fixed
9	Possible of reentrancy attack in `EcoERC20Pausable` contract	LOW	Fixed
10	Duplicated storage variables	LOW	Acknowledged
11	Incorrect receiver in `_completeUnstake`	INFORMATIVE	Acknowledged
12	Possible of incorrect value update for `bridgeTotalLocked`	INFORMATIVE	Fixed
13	The `_stakingInfo.ncpIds` array always have the length of one	INFORMATIVE	Acknowledged
14	Using both owner and admin for authorization is not recommended	INFORMATIVE	Acknowledged
15	Dynamic array and mapping shares the same storage slot	INFORMATIVE	Acknowledged
16	Incorrect error message in the `checkQuorum` function	INFORMATIVE	Fixed

2.2.1. Missing `bridgeTotalSyncStaking` update when invoking `toOriginSyncExtra` MEDIUM

Affected files:

- cross-bridge/CrossBridgeRemote.sol

In the `_processBridgeTransfer` function, the `localAsset.bridgeTotalSyncStaking` is not decreased when being invoked from the `toOriginSyncExtra`. However, this variable is increased in the `_processSyncReceive` function.

```
function toOriginSyncExtra(uint256 amount) whenNotPaused external payable override {
    // ...
    snapshot.localAsset = _processBridgeTransfer(snapshot, msgBridge, false);
```

```
    _updateAssetInfo(snapshot.localAsset);
}

function _processBridgeTransfer(
    ConfigSnapshot memory snapshot,
    ToOriginMessageInfo memory msgBridge,
    bool applyingFee
) internal returns(AssetInfo memory) {
    uint256 feeAmount;

    _checkBridgeTransferAmount(msgBridge.amount, snapshot.bridgeAmountConfig);
    if(applyingFee) {
        feeAmount = _calcFee(msgBridge.amount, snapshot.feeConfig);
        require(msgBridge.amount > feeAmount, "fee amount");
        unchecked{ msgBridge.amount -= feeAmount; }
        if(feeAmount != 0 )remoteToken.transfer(feeAccount, feeAmount);
    }
    snapshot.localAsset.bridgeTotalLocked -= uint128(msgBridge.amount);
    remoteToken.burn(msgBridge.amount);

    emit ToOrigin(_useLocalNonce(), msgBridge.account, msgBridge.amount, feeAmount);
    return snapshot.localAsset;
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.2. `setRemoteToken` will fail if the owner is changed MEDIUM

Affected files:

- cross-bridge/CrossBridgeRemote.sol

In the `initCrossBridgeRemote` function, the `setRemoteToken` is invoked after the `_transferOwnership`, which changes the owner address. Consequently, the `setRemoteToken` will fail due to the `onlyOwner` modifier.

```
function initCrossBridgeRemote(
    address owner,
    IEcoERC20 _remoteToken
) public initializer override {
    _transferOwnership(owner);
    setRemoteToken(_remoteToken);
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.3. `setStakingInfo` will fail if the owner is changed MEDIUM

Affected files:

- cross-bridge/CrossBridgeOrigin.sol

In the `initCrossBridgeOrigin` function, the `setStakingInfo` is invoked after the `_transferOwnership`, which changes the owner address. Consequently, the `setStakingInfo` will fail due to the `onlyOwner` modifier.

```
function initCrossBridgeOrigin(
    address owner,
    StakingInfo memory _stakingInfo
) public override initializer {
    _transferOwnership(owner);
    setStakingInfo(_stakingInfo);
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.4. Duplicated usage of staking value if `ncpIds.length > 1` MEDIUM

Affected files:

- cross-bridge/CrossBridgeOrigin.sol

In the `_stake` function, the `staking.deposit` is called multiple times if `ncpIds.length > 1`. However, the `amount` is used as `msg.value` for each call, which leads to the duplicated usage of the staking value.

```
function _stake(uint256 amount) internal {
    StakingInfo memory _stakingInfo = stakingInfo;
    INCPStaking staking = INCPStaking(_stakingInfo.stake_contract);
    (uint8[] memory ncpIds, uint128[] memory tmp) = _determineStakePid(_stakingInfo,
uint128(amount));
    for(uint256 i; i<ncpIds.length; i++) {
        staking.deposit{value: amount}(
            ncpIds[i], //ncpId default
            tmp[i], //amount
            payable(address(this)), //payable to
            false, //claimReward
            false //comp
        );
    }
}
```

RECOMMENDATION

Use the `tmp[i]` instead of `amount` as the `msg.value` for each call.

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.5. Incomplete implementation for `SupportUnderlyingNative` contract LOW

Affected files:

- stake/TokenizeStakeBase.sol

The `SupportUnderlyingNative` contract lacks implementation for `calcUnderlyingToWrap` and `calcWrapToUnderlying` functions. Furthermore, this contract is not currently used anywhere in the codebase, so we will consider it an incomplete implementation.

```
abstract contract SupportUnderlyingNative is ITokenizeStakeNative, TokenizeStakeBase {
    using SafeERC20 for IERC20Full;
    using Address for address payable;

    receive() external payable override {
        stakeValue(msg.sender);
    }

    function stakeValue(address account) public payable override returns (uint256
wrappedAmount) {
        require(msg.value != 0, "zero value");
        uint256 underlyingAmount = msg.value;
        wrappedAmount = calcUnderlyingToWrap(underlyingAmount);
        payable(address(underlying)).sendValue(underlyingAmount);
        _mint(account, wrappedAmount);
    }

    function unstakeValue(address account, uint256 wrappedAmount) public override returns
(uint256 underlyingAmount) {
        require(wrappedAmount != 0, "zero value");
        _burn(_msgSender(), wrappedAmount);
        underlyingAmount = calcWrapToUnderlying(wrappedAmount);
        IWETH(address(underlying)).withdraw(underlyingAmount);
        payable(account).sendValue(underlyingAmount);
    }
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team by removing the incomplete code.

2.2.6. Incomplete implementation for `SupportRewardNative` contract LOW

Affected files:

- stake/TokenizeStakeBase.sol

The `SupportRewardNative` contract is incomplete since the `_transferReward` is not called in either the `TokenizeStakeBase` or this contract. Additionally, it is not used anywhere in the codebase.

```
abstract contract SupportRewardNative is TokenizeStakeBase {
    using Address for address payable;

    function _transferReward(address account, uint256 amount) internal virtual override {
        if(amount != 0) payable(account).sendValue(amount);
    }
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team by removing the incomplete code.

2.2.7. Upgradeable contract but based on non-upgradeable ones LOW

Affected files:

- token/ERC20/EcoERC20.sol

The `EcoERC20` contract is designed to be upgradeable. However, it is built based on multiple non-upgradeable contracts such as `ERC20`, `SlotAdminable`, `Ownable`, etc. These contracts lack storage slot reservations, which are necessary for future upgrades.

```
contract SlotAdminable is
    IAdminable,
    Initializable,
    Ownable,
    SlotPausable,
    Multicall {}

abstract contract EcoERC20Mintable is IERC20Mintable, SlotAdminable, ERC20 {}

abstract contract EcoERC20Burnable is IERC20Burnable, ERC20Burnable {}

abstract contract EcoERC20MetadataInitializable is IERC20MetadataInitializable,
SlotAdminable, ERC20 {}

contract EcoERC20 is IEcoERC20, EcoERC20Mintable, EcoERC20Burnable,
EcoERC20MetadataInitializable {}
```

```
contract EcoERC20Pausable is IEcoERC20Pausable, EcoERC20 {}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team.

2.2.8. Incorrect initialization implementation for `EcoERC20Pausable` contract LOW

Affected files:

- token/ERC20/EcoERC20.sol

There are two issues in the `EcoERC20Pausable` contract.

Firstly, when deploying the `EcoERC20` contract, the `initEcoERC20` function is invoked twice, which is incorrect. The first invocation occurs in the `EcoERC20` constructor, and the second one is in the `initEcoERC20Pausable` call.

Secondly, these initialization functions should have the `internal` visibility and the `onlyInitializing` modifier to ensure that they are invoked once and during the initializing phase only. However, they are declared as `public` and with an incorrect modifier (`initializer`). The `initializer` modifier must only be used in the top-level initialization function.

```
contract EcoERC20 is IEcoERC20, EcoERC20Mintable, EcoERC20Burnable,
EcoERC20MetadataInitializable {
    constructor(
        address owner,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) ERC20("", "")
    {
        initEcoERC20(owner, name_, symbol_, decimals_);
    }

    function initEcoERC20(
        address owner,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) public initializer {
        initERC20Mintable(owner);
        initERC20MetadataInitializable(name_, symbol_, decimals_);
    }
    // ...
}

contract EcoERC20Pausable is IEcoERC20Pausable, EcoERC20 {
```

```
    constructor(
        address owner,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) EcoERC20(owner, name_, symbol_, decimals_) // Audit: initEcoERC20 (1)
    {
        initEcoERC20Pausable(owner, name_, symbol_, decimals_); // Audit: initEcoERC20 (1)
    }

    function initEcoERC20Pausable(
        address owner,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) public initializer {
        initEcoERC20(owner, name_, symbol_, decimals_);
    }
    // ...
}
```

RECOMMENDATION

The dev team should follow the OpenZeppelin's upgradeable contracts design pattern with the correct usage of `initializer` and `onlyInitializing` modifiers to ensure that each of the initialization functions are invoked only once.

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.9. Possible of reentrancy attack in `EcoERC20Pausable` contract LOW

Affected files:

- cross-bridge/CrossBridgeRemote.sol

In the `completeReceive` function, the call to `_trySendValueCatchFail` is not the last call. This may lead to the reentrancy attack if the `msgBridge.account` is a malicious contract. However, this vulnerability is still not exploitable in this case, so we consider it as a low severity issue.

```
// cross-bridge/CrossBridgeRemote.sol
function completeReceive(
    uint64 srcNonce, ToRemoteMessageInfo memory msgBridge, Sig[] memory sigs
) whenNotPaused external override {
    ConfigSnapshot memory snapshot = _getBridgeSnapshot();

    snapshot.localAsset = _processBridgeReceive(snapshot, srcNonce, msgBridge, sigs);
    _updateAssetInfo(snapshot.localAsset);
```

```
    _trySendValueCatchFail(payable(msgBridge.account),
snapshot.bridgeAmountConfig.toRemoteNativeSwapAmount);
    remoteToken.mint(msgBridge.account, msgBridge.amount - msgBridge.amountForNativeAlloc);
    if(msgBridge.amountForNativeAlloc != 0) remoteToken.mint(feeAccount,
msgBridge.amountForNativeAlloc);
}

// cross-bridge/CrossBridgeBase.sol
function _trySendValueCatchFail(
    address payable recipient, uint256 amount
) internal returns (bool success) {
    require(address(this).balance >= amount, "fail send value");

    (success, ) = recipient.call{ value: amount }(hex"");
    if (!success) {
        emit FailSendValue(recipient, amount);
        (bool _success, ) = payable(owner()).call{ value: amount }(hex"");
        require(_success, "fail amount catch");
    }
}
```

RECOMMENDATION

Move all of the `_trySendValueCatchFail` calls to the end of the function to ensure that no state changes occur afterward. Alternatively, use the `ReentrancyGuard` contract from OpenZeppelin to prevent reentrancy attacks.

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.10. Duplicated storage variables LOW

Affected files:

- token/ERC20/EcoERC20.sol

The `EcoERC20MetadataInitializable` contract inherits from the `ERC20` contract, which already contains the `_name` and `_symbol` state variables. However, these variables are redeclared in the `EcoERC20MetadataInitializable` contract, leading to unnecessary duplication of storage variables.

```
contract ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;
```

```
    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    // ...
}

abstract contract EcoERC20MetadataInitializable is IERC20MetadataInitializable,
SlotAdminable, ERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    // ...
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team.

2.2.11. Incorrect receiver in `_completeUnstake` INFORMATIVE

Affected files:

- cross-bridge/CrossBridgeOrigin.sol

In the `_completeUnstake` function, the `staking.withdraw()` is calledwith the `account` as receiver, set by the `_requestUnstake` function. However, the receiver must be set to `address(this)` so that the `CrossBridgeOrigin` contract can receive the withdrawn funds. In this case, this function will be reverted due to balance check after that.

```
function _completeUnstake(uint256 ncpId, uint256 withdrawalId) internal {
    address account = filterWithdrawalId[withdrawalId];
    require(account != address(0), "withdrawalId filter");
    delete filterWithdrawalId[withdrawalId];

    INCPStaking staking = INCPStaking(stakingInfo.stake_contract);
    uint256 amount = staking.withdrawalNFT().getWithdrawalRequestInfo(withdrawalId).amount;
    uint256 beforeBalance = address(this).balance;
    staking.withdraw(
        ncpId, // ncpId default
        withdrawalId, // withdrawal Id
        payable(account) // user address // Audit: Incorrect receiver
    );
    // Audit: function reverted by the following balance check
    require(address(this).balance - beforeBalance == amount, "withdraw amount");
    _trySendValueCatchFail(payable(account), amount);
    emit ReceiveComplete(account, ncpId, withdrawalId);
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team. However, this issue is due to the difference between the publicly disclosed codebase of WEMIX Foundation and the actual implementation of WONDER Staking contract.

2.2.12. Possible of incorrect value update for `bridgeTotalLocked` INFORMATIVE

Affected files:

- cross-bridge/CrossBridgeOrigin.sol

In the `_processBridgeTransfer` function, the `bridgeTotalLocked` is increased by a non-fee amount and the `bridgeTotalSyncStaking` is increased with fee. However, in the `sync` function, both the `bridgeTotalLocked` and `bridgeTotalSyncStaking` are increased. In conclusion, we suspect that the `bridgeTotalLocked` should be increased by the full amount, including the fee.

```
function _processBridgeTransfer(
    ConfigSnapshot memory snapshot,
    ToRemoteStartMessageInfo memory msgBridge,
    bool applyingFee
) internal returns(AssetInfo memory) {
    // ...
    if (applyingFee) {
        feeAmount = _calcFee(msgBridge.amount, snapshot.feeConfig);
        require(msgBridge.amount > feeAmount, "fee amount");
        unchecked{ msgBridge.amount -= feeAmount; }
        unchecked{ snapshot.localAsset.bridgeTotalSyncStaking += uint128(feeAmount); }
    }
    snapshot.localAsset.bridgeTotalLocked += uint128(msgBridge.amount);
    // ...
}

function sync() whenNotPaused external payable override {
    // ...
    snapshot.localAsset.bridgeTotalLocked += uint128(msg.value);
    snapshot.localAsset.bridgeTotalSyncStaking += uint128(msg.value);
    // ...
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

2.2.13. The `_stakingInfo.ncpIds` array always have the length of one INFORMATIVE

Affected files:

- cross-bridge/CrossBridgeBase.sol

In the `setStakingInfo` function, the length of `_ncpIds` is required to be equal one. Therefore, we should use a single value type to store it instead of an array.

```
function setStakingInfo(StakingInfo memory _stakingInfo) public override onlyOwner {
    require(_stakingInfo.stake_contract != address(0), "stake contract");
    require(_stakingInfo.ncpIds.length == 1, "ncpIds");

    stakingInfo = _stakingInfo;
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team.

2.2.14. Using both owner and admin for authorization is not recommended INFORMATIVE

Affected files:

- access/SlotAdminable.sol

In the `SlotAdminable` contract, the `onlyAuthorized` modifier is used to check if the caller is either the owner or the admin. However, it's better to use only one of them for authorization to avoid confusion and establish a clearer access control mechanism.

```
contract SlotAdminable is
    IAdminable,
    Initializable,
    Ownable,
    SlotPausable,
    Multicall
{
    // This is the keccak-256 hash of "adminable.address.mapping.slot" subtracted by 1
    bytes32 private constant _ADMIN_MAP_SLOT =
0xd4504e868494e8a2d3346e969ceecbe7706b48fa405166a42593e57599e9067b;
    modifier onlyAuthorized() {
        if (!(is_admin(_msgSender()) || _msgSender() == owner())) revert Adminship();
        _;
    }
    // ...
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team.

2.2.15. Dynamic array and mapping shares the same storage slot INFORMATIVE

Affected files:

- access/SlotServiceSigner.sol

In the `SlotServiceSigner` contract, the `_SIGNER_SLOT` is used to store both the `slotAddressArray` and `slotMappingToBoolean`. Currently, this will not produce any storage collision issues due to the storage layout of these dynamic types. However, we should use two separate slots for a clearer storage layout and to avoid potential issues in the future.

```
function _setSignerFlag(address signer, bool flag) internal {
    if (signer != address(0)) {
        if (_SIGNER_SLOT.slotMappingToBoolean()[signer] == flag) revert SignerAuth();
        _SIGNER_SLOT.slotMappingToBoolean()[signer] = flag;

        if(flag) _SIGNER_SLOT.slotAddressArray().push( signer );
        else _SIGNER_SLOT.slotAddressArray().remove( signer );
    }
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged by Kroma team.

2.2.16. Incorrect error message in the `checkQuorum` function INFORMATIVE

Affected files:

- access/SlotServiceSigner.sol

The error message in the `checkQuorum` function is incorrect. It should be `invalid signer` instead of `Ascending Order`.

```
function checkQuorum(address[] memory _signers) public view {
    uint256 len = _signers.length;
    // len >= total // 2 + 1
    require(len >= (_SIGNER_SLOT.slotAddressArray().length >> 1) + 1, "quorum");
    if(len > 1) {
        uint256 limit = len-1;
        for(uint256 i; i < limit;) {
            // avoid signature reuse
            require(_signers[i] < _signers[i+1], "Ascending Order");
            unchecked{ ++i; }
        }
    }

    mapping(address => bool) storage signerMap = _SIGNER_SLOT.slotMappingToBoolean();
    for(uint256 i; i < len;) {
        require(signerMap[ _signers[i] ], "Ascending Order"); // Audit: incorrect error
message
        unchecked{ ++i; }
    }
}
```

UPDATES

- *Jan 12, 2024*: This issue has been acknowledged and fixed by Kroma team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jan 12, 2024*	Public Report	Verichains Lab

Table 2. Report versions history