Bright Risk Index

Smart Contract Audit Report
Prepared for Bright Union

Date Issued:	Apr 12, 2022
Project ID:	AUDIT2022019
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2022019
Version	v1.0
Client	Bright Union
Project	Bright Risk Index
Auditor(s)	Peeraphut Punsuwan Darunphop Pengkumta Ronnachai chaipha Sorawish Laovakul
Author(s)	Peeraphut Punsuwan Natsasit Jirathammanuwat
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Apr 12, 2022	Full report	Peeraphut Punsuwan Natsasit Jirathammanuwat

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Bright Union, Inspex team conducted an audit to verify the security posture of the Bright Risk Index smart contracts between Mar 16, 2022 and Mar 22, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Bright Risk Index smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 critical, 3 high, 1 low, 1 very low, and 3 info-severity issues. With the project team's prompt response, 1 critical, 1 very low, and 2 info-severity issues were resolved and 3 high-severity were mitigated in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that Bright Risk Index smart contracts have sufficient protections to be safe for public use. However, as the source code is not publicly available, the bytecode of the smart contracts deployed should be compared with the bytecode of the smart contracts audited before interacting with them. In the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Bright Union is the platform that has a mission to make risk markets work for the crypto space. Bright Union is a collection of experts in crypto, technology, and insurance ready to bring web3.0 to the insurance industry.

Bright Risk Index is a decentralized insurance aggregator product that allows users to compare, buy, and provide coverage liquidity providers with the best experience. The liquid tokenized position (BRI) represents a curated basket of diversified staking positions underwriting risks in the DeFi insurance markets, creating a diversified portfolio, offering a novel and sound investment opportunity to profit by covering the community.

Scope Information:

Project Name	Bright Risk Index
Website	https://brightunion.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Ethereum Mainnet
Programming Language	Solidity
Category	Yield Aggregators

Audit Information:

Audit Method	Whitebox
Audit Date	Mar 16, 2022 - Mar 22, 2022
Reassessment Date	Apr 11, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The smart contracts with the following bytecodes were audited and reassessed by Inspex in detail:

Compiler version: v0.7.4+commit.3f05b770, optimization enabled with 200 runs.

Initial Audit:

Contract	Bytecode SHA256 Hash
BrightRiskToken	46c1d29aa0d6066e040752176a8b641961d86a2ed6f3208f8d1ba3c0ff438883
PriceFeed	844b99466710dc0d1c7ee09deba73d2f27648981458cee97c36c15a3373ae83d
AbstractController	-
InsuracePositionController	58a68c80f9b839d1954946440c848e14401e1bdce234b12733c8ade5e1a249c9
NexusPositionController	982a105d20cfc130c4a06b45f921734d49805c134ffff3e1c178692b317ae233
BridgeLeveragedPositionController	7e323914d7e61c7f3654760316b8954b71be367344e55a004007548a1dc7fc30

Reassessment Audit:

Contract	Bytecode SHA256 Hash
BrightRiskToken	0095f27a94ad65decacaed84515e3f6476cbff360f597064ea2c16e995e81497
PriceFeed	3349b65371ae9a93dcba610daf338b3626b5fd974cb10856f370b2a294686da3
AbstractController	-
InsuracePositionController	f1f854320dff48b3d6c0683d69fa75858aa0a7b318cb4acacb7cd6a527442052
NexusPositionController	3edc2ec3056962053ae55c2ac0f41e1598994d869dc5dd6e4b762131bcad553b
BridgeLeveragedPositionController	7086cdcd49c2275b66792e0d42b125ff47084e3fb23d652428384c2bd1826097

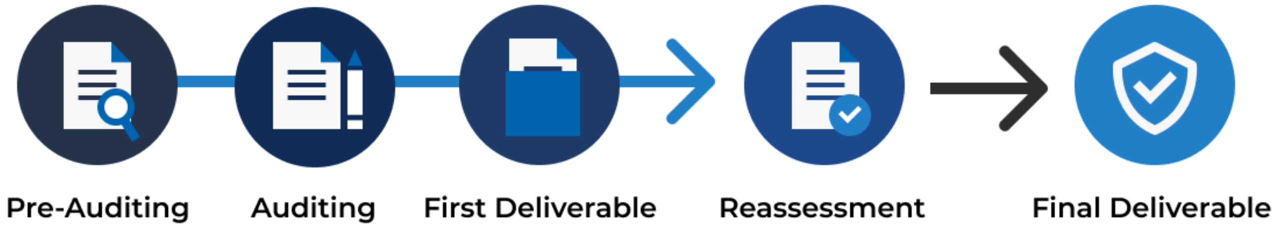
*AbstractController is an abstract contract that cannot be deployed

As the Bright Union team has decided not to publish the source code to protect their intellectual property, the users should compare the hashes of the bytecode hex string with the smart contracts deployed before interacting with them to make sure that they are the same with the contracts audited.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Smart Contract with Unpublished Source Code
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control

Broken Authentication
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

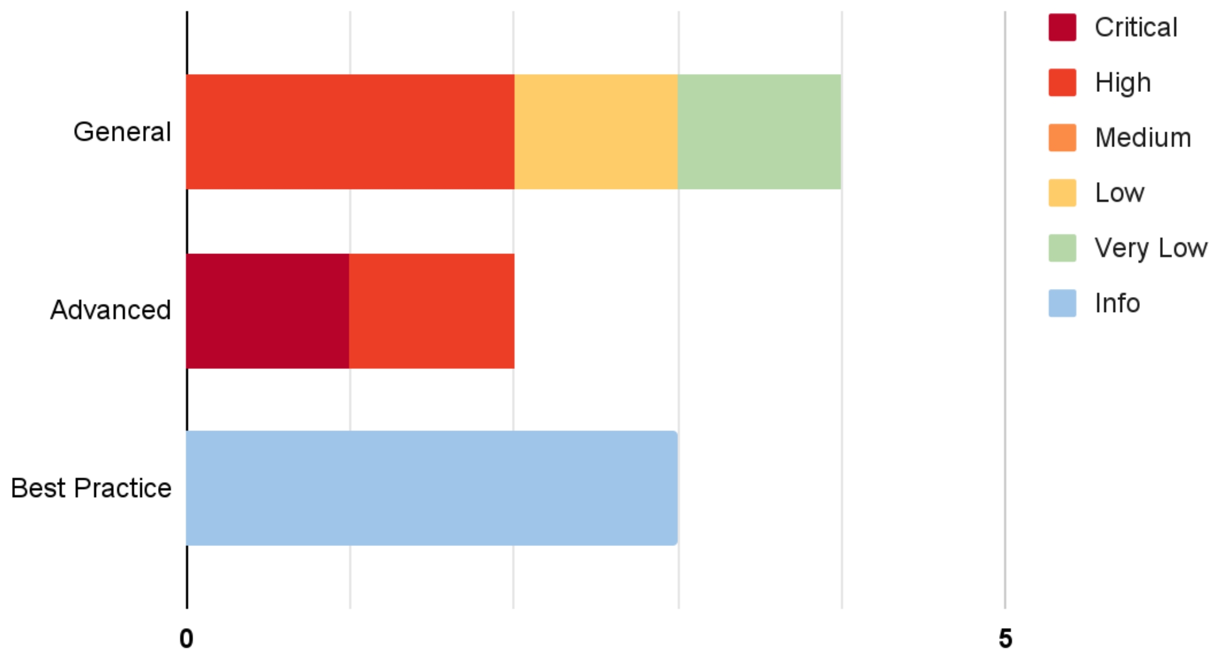
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

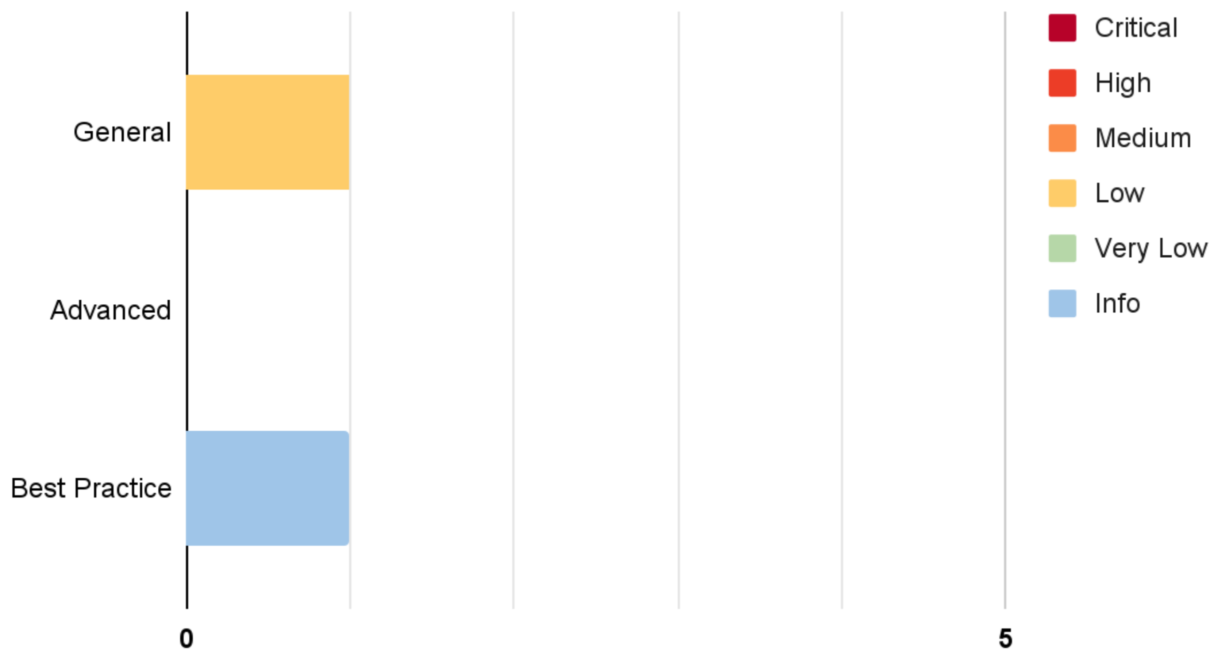
4. Summary of Findings

From the assessments, Inspex has found 9 issues in three categories. The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Denial of Services in Balance Checking	Advanced	Critical	Resolved
IDX-002	Arbitrary Price Setting	Advanced	High	Resolved *
IDX-003	Use of Upgradable Contract Design	General	High	Resolved *
IDX-004	Centralized Control of State Variable	General	High	Resolved *
IDX-005	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-006	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-007	Improper Function Visibility	Best Practice	Info	Resolved
IDX-008	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-009	Use of Duplicate Literals	Best Practice	Info	Resolved

* The mitigations or clarifications by Bright Union can be found in Chapter 5.

5. Detailed Findings Information

5.1. Denial of Services in Balance Checking

ID	IDX-001
Target	NexusPositionController InsuracePositionController
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity: Critical** **Impact: High** The platform admin won't be able to execute the core functions of the contract, causing disruption of service and loss of reputation to the platform. **Likelihood: High** It's easy to disrupt the service by transferring an amount of token (`$WNXM`, `$INSUR`) to the target contract.
Status	Resolved Bright Union team has resolved this issue by using the condition which checks whether the balance of the wallet is greater than the input amount value.

5.1.1. Description

The `NexusPositionController` contract allows the platform admin to stake funds into the Nexus platform through the `stake()` function.

The `stake()` function will buy a wrapped NXM token ($wNXM) and unwrap it to the NXM token ($NXM) after that the contract will check whether the $NXM balance of the contract itself is equal to the input amount or not.

NexusPositionController.sol

```
function stake(uint256 _amount) external override onlyIndex {
    require(canStake(), "NexusPositionController: NPC1");
    base.safeTransferFrom(_msgSender(), address(this), _amount);
    uint256 _wNXMAmount = _buyWNXM(_amount);
    require(_wNXMAmount != 0, "NexusPositionController: NPC2");
    require(_wNXMAmount >= _minStakingAmount(), "NexusPositionController: NPC3");

    _unwrapWNXM(_wNXMAmount);
    require(
```

```
84            _nxmToken.balanceOf(address(this)) == _wNXMAmount,
85            "NexusPositionController: NPC4"
86        );
87
88        uint256[] memory _stakes = _stakingStructure(_wNXMAmount);
89        _pooledStaking.depositAndStake(_wNXMAmount, allProducts, _stakes);
90
91        setStakingState();
92    }
```

According to the conditions at show in the following in the `NexusPositionController` contract at line 84, if the contract has already had some NXM tokens before, the condition result will always be `false`, causing every `stake()` transaction to be reverted.

Therefore, using == for checking the current token balance of the contract with the expected balance can cause a problem because anyone can transfer the token into the contract without any restriction.

In addition, the internal `_wrapNXM()` function in the `NexusPositionController` contract and the `withdrawRewards()` function in the `InsuracePositionController` contract that requires the contract's token balance to be equal to the expected balance may encounter the issues described above.

NexusPositionController.sol

```
function _wrapNXM(uint256 _amount) internal {
    _wNXMToken.wrap(_amount);
    require(_wNXMToken.balanceOf(address(this)) == _amount,
"NexusPositionController: NPC7");
}
```

InsuracePositionController.sol

```
function withdrawRewards() external override {
    (uint256 _withdrawableNow, ) = rewardsInVesting();

    (uint256 _rewards, uint256 _fee) = _applyFees(_withdrawableNow);

    _rewardController.withdrawReward(_withdrawableNow);

    require(
        ERC20(insurToken).balanceOf(address(this)) == _withdrawableNow,
        "InsuracePositionController IPC10"
    );

    //transfer fee in INSUR to Index
    ERC20(insurToken).transfer(feeInfo.feeRecipient, _fee);

    // swap INSUR into base
```

```
    _checkApprovals(IERC20(insurToken), address(_uniswapRouter), _rewards);
    uint256 _baseTokenAmount = _swapTokenForToken(
        _rewards,
        address(insurToken),
        address(base),
        swapRewardsVia
    );

    require(_baseTokenAmount != 0, "InsuracePositionController: IPC11");
    // Deposit to index
    index.depositInternal(_baseTokenAmount);
}
```

5.1.2. Remediation

Inspex suggests checking the current contract's token balance by using the >= or <= operators to avoid the issue when the user transfers the required token to the contract.

NexusPositionController.sol

```
function stake(uint256 _amount) external override onlyIndex {
    require(canStake(), "NexusPositionController: NPC1");
    base.safeTransferFrom(_msgSender(), address(this), _amount);
    uint256 _wNXMAmount = _buyWNXM(_amount);
    require(_wNXMAmount != 0, "NexusPositionController: NPC2");
    require(_wNXMAmount >= _minStakingAmount(), "NexusPositionController: NPC3");

    _unwrapWNXM(_wNXMAmount);
    require(
        _nxmToken.balanceOf(address(this)) >= _wNXMAmount,
        "NexusPositionController: NPC4"
    );

    uint256[] memory _stakes = _stakingStructure(_wNXMAmount);
    _pooledStaking.depositAndStake(_wNXMAmount, allProducts, _stakes);

    setStakingState();
}
```

NexusPositionController.sol

```
function _wrapNXM(uint256 _amount) internal {
    _wNXMToken.wrap(_amount);
    require(_wNXMToken.balanceOf(address(this)) >= _amount, "NexusPositionController: NPC7");
}
```

InsuracePositionController.sol

```
function withdrawRewards() external override {
    (uint256 _withdrawableNow, ) = rewardsInVesting();

    (uint256 _rewards, uint256 _fee) = _applyFees(_withdrawableNow);

    _rewardController.withdrawReward(_withdrawableNow);

    require(
        ERC20(insurToken).balanceOf(address(this)) >= _withdrawableNow,
        "InsuracePositionController IPC10"
    );

    //transfer fee in INSUR to Index
    ERC20(insurToken).transfer(feeInfo.feeRecipient, _fee);

    // swap INSUR into base
    _checkApprovals(IERC20(insurToken), address(_uniswapRouter), _rewards);
    uint256 _baseTokenAmount = _swapTokenForToken(
        _rewards,
        address(insurToken),
        address(base),
        swapRewardsVia
    );

    require(_baseTokenAmount != 0, "InsuracePositionController: IPC11");
    // Deposit to index
    index.depositInternal(_baseTokenAmount);
}
```

5.2. Arbitrary Price Setting

ID	IDX-002
Target	PriceFeed
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The price of assets can be manipulated by the owner to gain a profit when burning the `$BRI` to transfer the base token back. **Likelihood:** Medium Only the owner can set the price through the `setInternalPrice()` function.
Status	Resolved * Bright Union team has mitigated this issue by using the Chainlink price oracle as a data feed. However, there are some tokens that the Chailink does not provide a data feed. In this case, Bright Union team will set the price manually with a max price deviation is +/- 50% per update and the minimum update delay is set to 24 hrs.

5.2.1. Description

The `howManyTokensAinB()` function is used for calculating the amount that will be gotten from exchanging between two tokens based on the price of each token. There are two sources of the price that will be used in the function: the first one is obtaining the price from `Chainlink`, and the second one is obtaining from the `internalPriceFeed` state.

The owner can manipulate the result from the `howManyTokensAinB()` function by manipulating the price of a token that is being passed into the function.

PriceFeed.sol

```
function howManyTokensAinB(
    address tokenA,
    address tokenB,
    address via,
    uint256 amount,
    bool AMM
) external view override returns (uint256 _amountA) {
    //DEPRECATED, don't use Uniswap feed
    if (false) {
        _amountA = uniswapFeed(tokenA, tokenB, via, amount);
    } else {
```

```
        uint256 _priceA;
        if (chainlinkAggregators[tokenA] != address(0)) {
            (, int256 _price, , , ) =
AggregatorV3Interface(chainlinkAggregators[tokenA])
                .latestRoundData();
            _priceA = uint256(_price);
        } else if (internalPriceFeed[tokenA] != 0) {
            _priceA = internalPriceFeed[tokenA];
        }
        require(_priceA > 0, "PriceFeed: PF1");

        uint256 _priceB;
        if (chainlinkAggregators[tokenB] != address(0)) {
            (, int256 _price, , , ) =
AggregatorV3Interface(chainlinkAggregators[tokenB])
                .latestRoundData();
            _priceB = uint256(_price);
        } else if (internalPriceFeed[tokenB] != 0) {
            _priceB = internalPriceFeed[tokenB];
        }
        require(_priceB > 0, "PriceFeed: PF2");
        _amountA = _priceB.mul(amount).div(_priceA);

        uint256 _decimalsA = IERC20Internal(tokenA).decimals();
        uint256 _decimalsB = IERC20Internal(tokenB).decimals();
        if (_decimalsA > _decimalsB) {
            _amountA = _amountA.mul(10(_decimalsA - _decimalsB));
        } else if (_decimalsB > _decimalsA) {
            _amountA = _amountA.div(10(_decimalsB - _decimalsA));
        }
    }
}
```

To manipulate, firstly, the owner has to set the `chainlinkAggregators` state of the token with the value of address 0 through the `addChainlinkAggregator()` function; this will force the function to use the price from the `internalPriceFeed` state instead.

PriceFeed.sol

```
function addChainlinkAggregator(address _token, address _aggregator) external
onlyOwner {
    chainlinkAggregators[_token] = _aggregator;
}
```

Secondly, the owner can arbitrarily control the price of the token through the `setInternalPrice()` function. The `internalPriceFeed` state internally stores the price of the tokens that will be used in the `howManyTokensAinB()` function if it does not use the price from `Chainlink`.

PriceFeed.sol

```
function setInternalPrice(address _token, uint256 _price) external onlyOwner {
    internalPriceFeed[_token] = _price;
}
```

The result from the `howManyTokensAinB()` function has effects on the amount of `$BRI` that the user will receive from depositing the capital or the amount of the capital that will be received from burning `$BRI`.

For example, to manipulate the amount that will get from burning the index token, `$BRI`, the amount of the burned token will be passed to the `convertIndexToInvestment()` function to calculate the amount of the capital from burning the token.

BrightRiskToken.sol

```
function burn(uint256 _indexTokenAmount) external whenNotPaused {
    require(_indexTokenAmount > 0, "BrightRiskToken: BRI15");
    require(balanceOf(_msgSender()) >= _indexTokenAmount, "BrightRiskToken: BRI16");

    uint256 _investments = convertIndexToInvestment(_indexTokenAmount);
    // apply the fee
    _investments = _applyDepositFee(_investments);
    require(externalPool >= _investments, "BrightRiskToken: BRI17");

    _burn(_msgSender(), _indexTokenAmount);
    externalPool = externalPool.sub(_investments);
    base.transfer(_msgSender(), _investments);
    emit IndexBurn(_msgSender(), _indexTokenAmount, _investments);
}
```

The result from the `convertIndexToInvestment()` function depends on the value from the `_indexRatio()` function.

BrightRiskToken.sol

```
function convertIndexToInvestment(uint256 _amount) public view returns (uint256) {
    return _amount.mul(_indexRatio()).div(PERCENTAGE_100);
}
```

The result from `_indexRatio()` function depends on the results from the `totalSupply()` function and the `totalTVL()` function. The result from the `totalSupply()` function is the amount of `$BRI`, and the result from the `totalTVL()` function is the total value of the staking assets.

BrightRiskToken.sol

```
function _indexRatio() internal view returns (uint256 _ratio) {
```

```
    uint256 _stakes = totalTVL();
    uint256 _currentTotalSupply = totalSupply();

    if (_stakes == 0 || _currentTotalSupply == 0) {
        _ratio = PERCENTAGE_100;
    } else {
        _ratio = _stakes.mul(PRECISION).div(_currentTotalSupply);
    }
    _ratio = _ratio.mul(100); //factor x100
}
```

The total value of the staking assets from the `totalTVL()` function is the sum of the result from the `netWorth()` function of each staking position with the `externalPool` state.

BrightRiskToken.sol

```
function totalTVL() public view returns (uint256 _tvl) {
    uint256 _to = _positionControllers.length();
    for (uint256 i = 0; i < _to; i++) {
        _tvl =
_tvl.add(IPositionController(_positionControllers.at(i)).netWorth());
    }
    _tvl = _tvl.add(externalPool);
}
```

The function `netWorth()` of the `IPositionController` interface can be implemented differently, but all of them(`NexusPositionController`,`InsuracePositionController`,`BridgeLeveragedPositionController`) depends on the result from the `howManyTokensAinB()` function of the `PriceFeed` contract.

NexusPositionController.sol

```
function netWorth() external view override returns (uint256) {
    (uint256 _rewards, ) = _calculateRewards();
    //how many DAI in wNXM
    return
        _priceFeed.howManyTokensAinB(
            address(base),
            address(_wNXMToken),
            swapRewardsVia,
            _deposit().add(_rewards),
            true
        );
}
```

InsuracePositionController.sol

```
function netWorth() external view override returns (uint256) {
    (uint256 _rewards, ) = _calculateRewards();

    uint256 _rewardsBase;
    if (_rewards != 0) {
        // returns INSUR rewards in base asset
        _rewardsBase = _priceFeed.howManyTokensAinB(
            address(base),
            insurToken,
            swapRewardsVia,
            _rewards,
            true
        );
    }

    // returns currently staked amount in base asset
    uint256 _stakedBase;
    if (address(base) != stakingAsset) {
        _stakedBase = _priceFeed.howManyTokensAinB(
            address(base),
            stakingAsset,
            swapVia,
            positionSupply(),
            true
        );
    } else {
        _stakedBase = positionSupply();
    }

    return _stakedBase.add(_rewardsBase);
}
```

BridgeLeveragedPositionController.sol

```
function netWorth() external view override returns (uint256 _worth) {
    //staking
    uint256 _stakedStbl;
    uint256 _rewardsBase;
    (uint256 _bmiRewards, ) = _calculateRewards();
    if (_bmiRewards > 0) {
        _rewardsBase = _priceFeed.howManyTokensAinB(
            address(base),
            address(bmiToken),
            swapRewardsVia,
            _bmiRewards,
            true
```

```
            );
        }
        //'deposited and then staked'
        _stakedStbl = bmiCoverStaking.totalStakedSTBL(address(this));
        (uint256 _unstakingBMIX, ) = unstakingInfo();
        //only 'deposited'
        _stakedStbl =
_stakedStbl.add(leveragedPortfolio.convertBMIXToSTBL(_unstakingBMIX));

        _stakedStbl = DecimalsConverter.convertFrom18(_stakedStbl, stblDecimals);
        _worth = _priceFeed
            .howManyTokensAinB(address(base), address(stblToken), swapVia,
_stakedStbl, true)
            .add(_rewardsBase);
}
```

So, the owner can manipulate prices in the `internalPriceFeed` state to be an absurdly high value or the value that could result in the burning token will exactly yield the whole deposited capital. The chain of effect can be crudely summarized as follows:

`internalPriceFeed` → `howManyTokensAinB()` → `netWorth()` → `totalTVL()` → `_indexRatio()` → `convertIndexToInvestment()` → `burn()`

5.2.2. Remediation

Inspex suggests removing the `setInternalPrice()` function. The token price should not be set manually. Inspex suggests using the price data from a trustable price oracle provider.

If the price of the needed tokens is not available from other trustable sources, Inspex suggests using the time-weight average price (TWAP Oracle) instead of directly quoting from the reserves (https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles).

5.3. Use of Upgradable Contract Design

ID	IDX-003
Target	BrightRiskToken PriceFeed AbstractController InsuracePositionController NexusPositionController BridgeLeveragedPositionController
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of the affected contract can be arbitrarily changed. This allows the proxy owner to perform malicious actions, e.g., stealing the user funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	Resolved * Bright Union team has confirmed that the AdminProxy contract which owns all upgradable contracts will be transferred its ownership to the `Timelock` contract. However, as the affected contracts are not transferred the ownership during the reassessment, the users should confirm that the contracts are under the effect of the `Timelock` contract before using them.

5.3.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As the smart contracts are upgradable, the contract logic can be modified by the owner anytime, making the smart contract untrustworthy.

5.3.2. Remediation

Inspex suggests deploying the contract without the proxy pattern or any solution that can make the smart contract upgradable.

However, if upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes. This allows the platform users to monitor the timelock and is notified of the potential changes being done on the smart contract.

5.4. Centralized Control of State Variable

ID	IDX-004
Target	BrightRiskToken PriceFeed
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles.
Status	Resolved * Bright Union team has confirmed that the privilege functions will be called through the `Timelock` contract. This means any action that would occur to the privilege function will be able to be monitored by the community conveniently. However, as the affected contracts are not transferred the ownership during the reassessment, the users should confirm that the contracts are under the effect of the `Timelock` contract before using them.

5.4.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
BrightRiskToken.sol (L:191)	addController()	onlyOperator
BrightRiskToken.sol (L:198)	removeController()	onlyOperator
BrightRiskToken.sol (L:206)	setMinimumDeposit()	onlyOperator

BrightRiskToken.sol (L:212)	setDepositFee()	onlyOperator
BrightRiskToken.sol (L:219)	adjustStreamingFee()	onlyOperator
BrightRiskToken.sol (L:237)	setSwapViaAt()	onlyOperator
BrightRiskToken.sol (L:243)	setSwapRewardsViaAt()	onlyOperator
BrightRiskToken.sol (L:314)	setPriceFeed()	onlyAdmin
PriceFeed.sol (L:115)	addChainlinkAggregator()	onlyOwner
PriceFeed.sol (L:123)	setInternalPrice()	onlyOwner

5.4.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours.

However, if the timelock is used in the `BrightRiskToken` contract, the role of the `whenNotPaused` modifier should be changed from the `operator` role to another role to prevent the emergency functions (`pause()` and `unpause()` functions) being stuck in the timelock, in case of emergency use.

5.5. Smart Contract with Unpublished Source Code

ID	IDX-005
Target	BrightRiskToken PriceFeed AbstractController InsuracePositionController NexusPositionController BridgeLeveragedPositionController
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract. **Likelihood:** Low The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.
Status	**Acknowledged** Bright Union team has acknowledged this issue and decided not to publish the source code at this moment to protect the intellectual property.

5.5.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

5.5.2. Remediation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

5.6. Insufficient Logging for Privileged Functions

ID	IDX-006
Target	BrightRiskToken PriceFeed AbstractController InsuracePositionController NexusPositionController BridgeLeveragedPositionController
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved Bright Union team has resolved this issue as suggested by emitting events for all privilege functions.

5.6.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For Example, the operator could set the deposit fee by executing the `setDepositFee()` function in the `BrightRiskToken` contract, and no events are emitted.

BrightRiskToken.sol

```
function setDepositFee(uint256 _fee) external onlyOperator {
    require(_fee < 5 * 10**5, "BrightRiskToken: BRI25");
    depositFee = _fee;
}
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
BrightRiskToken.sol (L:191)	BrightRiskToken	addController()
BrightRiskToken.sol (L:198)	BrightRiskToken	removeController()
BrightRiskToken.sol (L:206)	BrightRiskToken	setMinimumDeposit()
BrightRiskToken.sol (L:212)	BrightRiskToken	setDepositFee()
BrightRiskToken.sol (L:219)	BrightRiskToken	adjustStreamingFee()
BrightRiskToken.sol (L:237)	BrightRiskToken	setSwapViaAt()
BrightRiskToken.sol (L:243)	BrightRiskToken	setSwapRewardsViaAt()
BrightRiskToken.sol (L:314)	BrightRiskToken	setPriceFeed()
PriceFeed.sol (L:115)	PriceFeed	addChainlinkAggregator()
PriceFeed.sol (L:123)	PriceFeed	setInternalPrice()
AbstractController.sol (L:75)	AbstractController	setFeeInfo()
AbstractController.sol (L:79)	AbstractController	setSwapVia()
AbstractController.sol (L:83)	AbstractController	setSwapRewardsVia()
NexusPositionController.sol (L:94)	NexusPositionController	callUnstake()
NexusPositionController.sol (L:104)	NexusPositionController	unstake()
InsuracePositionController.sol (L:111)	InsuracePositionController	callUnstake()
InsuracePositionController.sol (L:120)	InsuracePositionController	unstake()
BridgeLeveragedPositionController.sol (L:143)	BridgeLeveragedPositionController	unstake()
BridgeLeveragedPositionController.sol (L:167)	BridgeLeveragedPositionController	setActiveNftId()

5.6.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

BrightRiskToken.sol

```
event setDepositFree(uint256 _fee);
function setDepositFee(uint256 _fee) external onlyOperator {
    require(_fee < 5 * 10**5, "BrightRiskToken: BRI25");
    depositFee = _fee;
    emit setDepositeFee(_fee);
}
```

5.7. Improper Function Visibility

ID	IDX-007
Target	AbstractController
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Bright Union team has resolved this issue as suggested by changing functions' visibility to external.

5.7.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `setFeeInfo()` function on the `AbstractController` contact is set to public and it is never called from any internal function.

AbstractController.sol

```
75 function setFeeInfo(FeeInfo memory _feeInfo) public onlyOwner {
76     feeInfo = _feeInfo;
77 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

Target	Function
AbstractController.sol (L:75)	setFeeInfo()
AbstractController.sol (L:79)	setSwapVia()
AbstractController.sol (L:83)	setSwapRewardsVia()

5.7.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function, as shown in the following example:

AbstractController.sol

```
function setFeeInfo(FeeInfo memory _feeInfo) external onlyOwner {
    feeInfo = _feeInfo;
}
```

5.8. Inexplicit Solidity Compiler Version

ID	IDX-008
Target	BrightRiskToken PriceFeed AbstractController NexusPositionController InsuracePositionController BridgeLeveragedPositionController
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact

5.8.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

BrightRiskToken.sol

```
pragma solidity ^0.7.4;
```

The following table contains all targets which the inexplicit compiler version is declared.

Contract	Version
BrightRiskToken	^0.7.4
PriceFeed	^0.7.4
AbstractController	^0.7.4
NexusPositionController	^0.7.4
InsuracePositionController	^0.7.4
BridgeLeveragedPositionController	^0.7.4

5.8.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.7 is v0.7.6 (https://docs.soliditylang.org/en/v0.7.6/).

BrightRiskToken.sol

```
pragma solidity 0.7.6;
```

5.9. Use of Duplicate Literals

ID	IDX-009
Target	BrightRiskToken
Category	Smart Contract Best Practice
CWE	CWE-1106: Insufficient Use of Symbolic Constants
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Bright Union team has resolved this issue as suggested by using `FEE_PRECISION` and `FEE_PERCENTAGE_100` constants.

5.9.1. Description

The use of literal numbers in the calculations can reduce the readiness of the contract. There are literal numbers in the `BrightRiskToken` contract that refer to the same value, i.e., the precision of the `depositFee` state. These numbers can be defined under the same constant variable to make the functions more readable and improve maintainability.

BrightRiskToken.sol

```
function setDepositFee(uint256 _fee) external onlyOperator {
    require(_fee < 5 * 10**5, "BrightRiskToken: BRI25");
    depositFee = _fee;
}
```

BrightRiskToken.sol

```
function _applyDepositFee(uint256 _amount) internal view returns (uint256
_withFee) {
    _withFee = _amount.mul(100 * 105 - depositFee).div(100 * 105);
}
```

5.9.2. Remediation

Inspex suggests declaring constants for the value of the fee's literals and using them in the `setDepositFee()` function and the `_applyDepositFee()` function. For example, declare the `FEE_PRECISION` constant for the value of the fee's precision and the `FEE_PERCENTAGE_100` constant for the value of the fee's percentage at one hundred percent.

BrightRiskToken.sol

```
uint256 constant FEE_PRECISION = 10**5;
uint256 constant FEE_PERCENTAGE_100 = 100 * FEE_PRECISION;
```

BrightRiskToken.sol

```
function setDepositFee(uint256 _fee) external onlyOperator {
    require(_fee < 5 * FEE_PRECISION, "BrightRiskToken: BRI25");
    depositFee = _fee;
}
```

BrightRiskToken.sol

```
function _applyDepositFee(uint256 _amount) internal view returns (uint256
_withFee) {
    _withFee = _amount.mul(FEE_PERCENTAGE_100 -
depositFee).div(FEE_PERCENTAGE_100);
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE