ybBlast

Smart Contract Audit Report
Prepared for HMX

Date Issued:	Mar 1, 2024
Project ID:	AUDIT2024002
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2024002
Version	v1.0
Client	HMX
Project	ybBlast
Auditor(s)	Wachirawit Kanpanluk Ronnachai Chaipha
Author(s)	Wachirawit Kanpanluk
Reviewer	Peeraphut Punsuwan
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Mar 1, 2024	Full report	Wachirawit Kanpanluk

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 HMX, Inspex team conducted an audit to verify the security posture of the ybBlast smart contracts on Feb 22, 2024. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of ybBlast 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 medium and 1 very low-severity issues. With the project team's prompt response, 1 medium-severity issue was resolved in the reassessment, while 1 very low-severity issue was acknowledged by the team. Therefore, Inspex trusts that ybBlast smart contracts have sufficient protections to be safe for public use. However, 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

The ybETH and ybUSDB contracts are designed to wrap auto-rebasing tokens on Blast, operating similarly to wstETH. The balances of all yield-bearing tokens will remain static, while the conversion rate to the actual underlying asset will continuously increase.

Scope Information:

Project Name	ybBlast
Website	https://hmx.org
Smart Contract Type	Ethereum Smart Contract
Chain	Blast
Programming Language	Solidity
Category	Token

Audit Information:

Audit Method	Whitebox
Audit Date	Feb 22, 2024
Reassessment Date	Feb 29, 2024

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 following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 978d8d0676e17009eb39de01c46c9a7ba7faba7e)

Contract	Location (URL)
ybETH	https://github.com/HMXOrg/yb-blast/blob/978d8d0676/src/ybETH.sol
ybUSDB	https://github.com/HMXOrg/yb-blast/blob/978d8d0676/src/ybUSDB.sol

Reassessment: (Commit: 6081399057cd47dbcfdcbbe8dfacd4007f8c66a4)

Contract	Location (URL)
ybETH	https://github.com/HMXOrg/yb-blast/blob/6081399057/src/ybETH.sol
ybUSDB	https://github.com/HMXOrg/yb-blast/blob/6081399057/src/ybUSDB.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

2.3. Security Model

2.3.1 Trust Modules

Several functionalities in the `ybETH` and `ybUSDB` contracts have relied on the external components, which may significantly impact the contract if they malfunction. The external components are listed as follows:

- The `WETHRebasing` and `USDB` contracts that the platform is interacting with to stake assets and claim yields to users; this operation is included in the deposit and withdrawal process; a failure of contracts may lead to denial of services.

2.3.2 Trust Assumptions

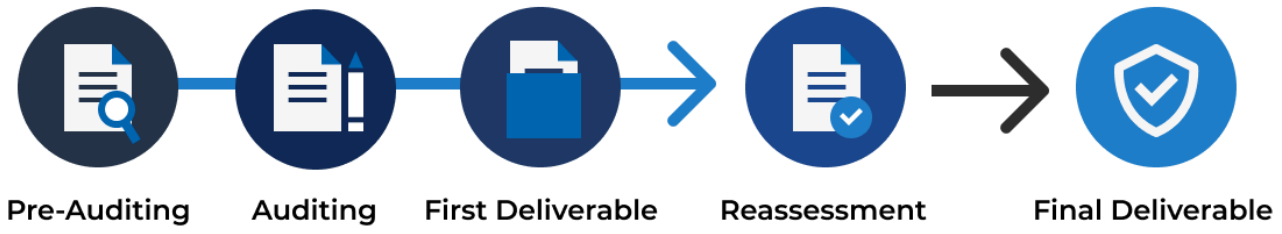
In the `ybETH` and `ybUSDB` contracts, the protocol's external components were assumed to be trusted. Acknowledging these trust assumptions is important, as it introduces substantial risks to the platform. Trust assumptions include, but are not limited to:

- The `WETHRebasing` and `USDB` tokens are assumed to work correctly all the time.
- The Blast's yield reporter reports the yield to the `WETHRebasing` and `USDB` tokens correctly.

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 testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at (https://docs.inspex.co/smart-contract-security-testing-guide/).

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

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**.

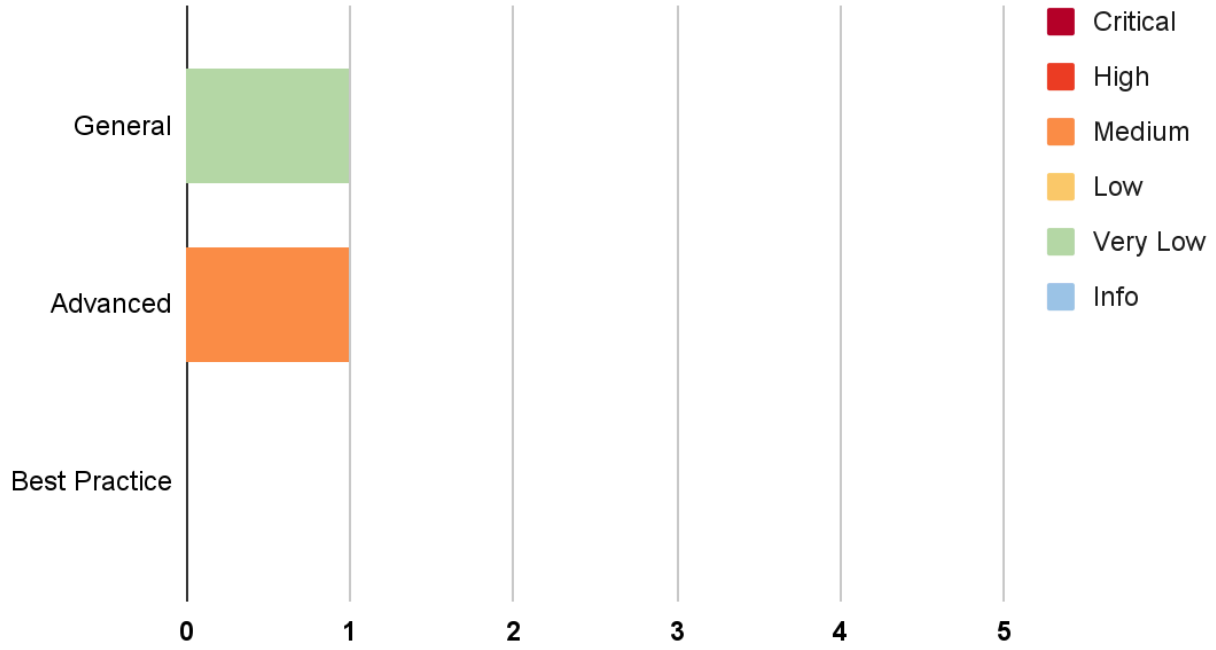
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

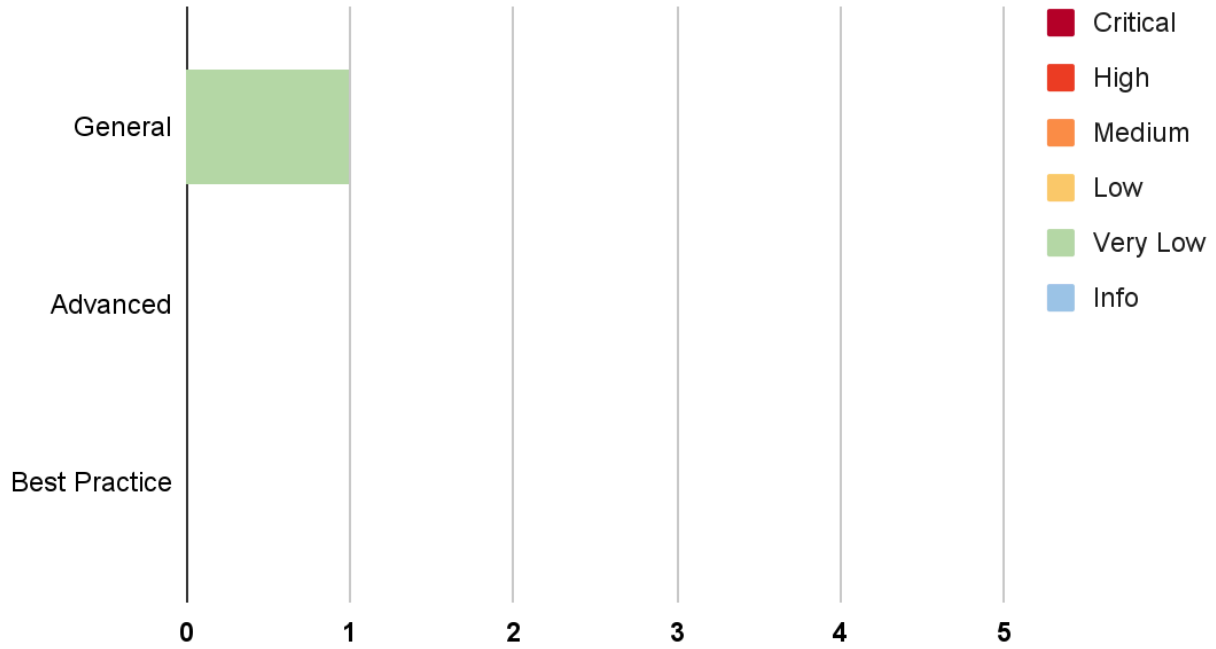
4. Summary of Findings

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	Share Value Inflation Attacks	Advanced	Medium	Resolved
IDX-002	Outdated Compiler Version	General	Very Low	Acknowledged

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

5. Detailed Findings Information

5.1. Share Value Inflation Attacks

ID	IDX-001
Target	ybETH ybUSDB
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Medium **Impact:** High The share value can be inflated due to the rounding, which takes some assets from the user in favor of the vault. This could allow the attacker to steal the other users' assets while depositing on the platform. **Likelihood:** Low In order to gain a profit, the attacker has to be the first depositor to fully control the share value, while the Blast's yield reporter is reporting to the rebasing token contract.
Status	Resolved The team has resolved this issue by minting the dead share to the contract, as suggested in commit `6012a87756286d699fd358a6a423c430f908ec82`.

5.1.1. Description

When depositing and withdrawing, the number of users' shares is rounded to favor the platform. This rounding takes some assets from the user to the contract, which allows the attacker to perform the donation attack to inflate the share value of the `ybETH` and `ybUSDB` tokens.

In the `withdraw()` and `deposit()` functions implemented within the `ybETH` and `ybUSDB` contracts, there is a mechanism that rounds the division results.

For example, in the `withdraw()` function, the `_shares` to be burned are calculated in the `previewWithdraw()` function in line 143. The share is calculated by using the `mulDivUp()` function, which rounds up the number of shares to burn.

ybUSDB.sol

```
139 function withdraw(uint256 _assets, address _receiver, address _owner) public
    returns (uint256 _shares) {
140   // Claim all pending yield
141   claimAllYield();
142
```

```
143     _shares = previewWithdraw(_assets);
144
145     if (msg.sender != _owner) {
146       // If msg.sender is not the owner, then check allowance
147       uint256 _allowed = allowance[_owner][msg.sender];
148       if (_allowed != type(uint256).max) {
149         // If not unlimited allowance, then decrease allowance.
150         // This should be reverted if the allowance is not enough.
151         allowance[_owner][msg.sender] = _allowed - _shares;
152       }
153     }
154
155     // Effect
156     _burn(_owner, _shares);
157     _totalAssets -= _assets;
158
159     // Interaction
160     // Transfer assets out
161     asset.safeTransfer(_receiver, _assets);
162
163     emit Withdraw(msg.sender, _receiver, _owner, _assets, _shares);
164 }
```

ybUSDB.sol

```
194 function previewWithdraw(uint256 _assets) public view returns (uint256 _shares)
    {
195     // SLOAD
196     uint256 _totalSupply = totalSupply;
197     return _totalSupply == 0 ? _assets : _assets.mulDivUp(_totalSupply,
    totalAssets());
198 }
```

Consider a scenario where the platform receives a yield from the Blast's rebasing token contract, altering the share-to-asset ratio. For instance, with a 100:101 share-to-asset ratio, withdrawing 101 assets normally results in the burning of 100 shares.

However, if an attacker withdraws 1 asset 99 times consecutively, the rounding up logic in the `previewWithdraw()` function will lead to the burning of 99 shares in total. This will result in an inflation of share value since the share-to-asset ratio is currently 1:2.

Similarly, in the `deposit()` function, the share value can also be manipulated by increasing the number of deposited assets while the number of minted shares rises less due to being rounded down with the `previewDeposit()` function.

ybUSDB.sol

```
59  function deposit(uint256 _assets, address _receiver) external returns (uint256
    _shares) {
60    // Claim all pending yield
61    claimAllYield();
62
63    // Check for rounding error.
64    if ((_shares = previewDeposit(_assets)) == 0) revert ZeroShares();
65
66    // Transfer from depositor
67    asset.safeTransferFrom(msg.sender, address(this), _assets);
68
69    // Effect
70    // Update totalAssets
71    _totalAssets += _assets;
72    // Mint ybUSDB
73    _mint(_receiver, _shares);
74
75    // Log
76    emit Deposit(msg.sender, _receiver, _assets, _shares);
77  }
```

ybUSDB.sol

```
174 function previewDeposit(uint256 _assets) public view returns (uint256 _shares)
    {
175   return convertToShares(_assets);
176 }
```

ybUSDB.sol

```
202 function convertToShares(uint256 _assets) public view returns (uint256 _shares)
    {
203   // SLOAD
204   uint256 _totalSupply = totalSupply;
205   return _totalSupply == 0 ? _assets : _assets.mulDivDown(_totalSupply,
    totalAssets());
206 }
```

For example, under a 1:2 share-to-asset ratio, a typical deposit of 4 assets would normally lead to an increase of 2 in the total supply.

However, if an attacker deposits 3 assets, the `convertToShares()` function might round down the result to 1 share, resulting in the total assets increasing by 3 while the total supply only increases by 1. This will cause an inflation in the share value since the current share-to-asset ratio will be 2:5.

The attacker can repeatedly deposit and withdraw to decrease the total share and increase the total asset.

This results in a share value that can be infinitely inflated by the attacker.

5.1.2. Remediation

Inspex suggests mitigating the issue by adding the dead shares as an initial asset deposit after deployment, for example:

ybUSDB.sol

```
37  constructor(IERC20Rebasing _usdb, uint256 _assets) ERC20("ybUSDB", "ybUSDB",
    18) {
38      // Effect
39      asset = _usdb;
40      yieldInbox = new YieldInbox();
41
42      asset.configure(YieldMode.CLAIMABLE);
43
44      // add dead shares, assume 1 _asset = 1 share
45      _totalAssets += _assets;
46      _mint(address(this), _assets);
47  }
```

The dead shares absorb excess tokens and counterbalances any inflationary pressures on the share's value. Its effectiveness varies depending on the quantity of the dead shares minted to the contract.

Alternatively, mitigation can be achieved by adding virtual shares and decimal offsets to the share value conversion and preview functions.

ybUSDB.sol

```
37  constructor(IERC20Rebasing _usdb) ERC20("ybUSDB", "ybUSDB", 18 +
    uint8(_decimalOffset)) {
38      // Effect
39      asset = _usdb;
40      yieldInbox = new YieldInbox();
41
42      asset.configure(YieldMode.CLAIMABLE);
43  }
```

ybUSDB.sol

```
186 function previewMint(uint256 _shares) public view returns (uint256 _assets) {
187     // SLOAD
188     uint256 _totalSupply = totalSupply;
189     return _shares.mulDivUp(totalAssets() + 1, _totalSupply +
    10_decimalOffset);
190   }
191
192 /// @notice Preview the amount of ybUSDB needed by specifying the amount of
```

```
    USDB wishes to receive.
193 /// @param _assets The amount of USDB wishes to receive.
194 function previewWithdraw(uint256 _assets) public view returns (uint256 _shares)
    {
195     // SLOAD
196     uint256 _totalSupply = totalSupply;
197     return  _assets.mulDivUp(_totalSupply + 10**_decimalOffset, totalAssets() +
    1);
198 }
199
200 /// @notice Convert the amount of assets to ybUSDB.
201 /// @param _assets The amount of assets to convert.
202 function convertToShares(uint256 _assets) public view returns (uint256 _shares)
    {
203     // SLOAD
204     uint256 _totalSupply = totalSupply;
205     return  _assets.mulDivDown(_totalSupply + 10**_decimalOffset, totalAssets()
    + 1);
206 }
207
208 /// @notice Convert the amount of ybUSDB to assets.
209 /// @param _shares The amount of ybUSDB to convert.
210 function convertToAssets(uint256 _shares) public view returns (uint256 _assets)
    {
211     // SLOAD
212     uint256 _totalSupply = totalSupply;
213     return  _shares.mulDivDown(totalAssets() + 1, _totalSupply +
    10_decimalOffset);
214 }
```

The `_decimalOffset` state increases precision and reduces rounding errors in share calculations. Additionally, the virtual shares, represented by `totalAssets() + 1`, act as the dead share in the contract, absorbing manipulated balances and mitigating their impact on the system when there is no depositor.

Together, these prevent inflation attacks by minimizing profitability and increasing the portion of the donation to the attacker depending on the value of the `_decimalOffset` state.

More information can be found:

- https://docs.openzeppelin.com/contracts/4.x/erc4626
- https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks

5.2. Outdated Compiler Version

ID	IDX-002
Target	ybETH ybUSDB
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Acknowledged** The team has acknowledged this issue. However, the outdated compiler version has no direct impact.

5.2.1. Description

The Solidity compiler versions specified in the smart contracts were outdated (https://docs.soliditylang.org/en/latest/bugs.html). As the compilers are regularly updated with bug fixes and new features, the latest stable compiler version should be used to compile the smart contracts for best practice.

ybUSDB.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.19;
```

The table below represents the contracts that apply the outdated Solidity compiler version.

File	Version
ybETH.sol (L:2)	0.8.19
ybUSDB.sol (L:2)	0.8.19

5.2.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version (https://github.com/ethereum/solidity/releases). At the time of audit, the latest stable version of the Solidity compiler in major 0.8 is 0.8.24.

For chains that may not be compatible with Solidity compiler version 0.8.24, Inspex suggests using Solidity compiler version 0.8.19 instead, as Solidity compiler version 0.8.20 or later introduces the PUSH0 (0x5f) opcode, which some chains have not yet included. (https://github.com/ethereum/solidity/releases/tag/v0.8.20)

ybUSDB.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.24;
```

6. Appendix

6.1. About Inspex

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