



CERTORA

Formal Verification of YieldBox (Tapioca) - Appendix

Summary

This document describes the specification and verification of Tapioca / BoringCrypto's contract YieldBox using the Certora Prover. The latest commit that was reviewed and run through the Certora Prover was [d5908966](#).

The scope of this verification is Tapioca / BoringCrypto's contract `YieldBox.sol`. No default strategy was provided as part of the scope. The strategy's usage in YieldBox is crucial to mitigating some of the issues.

The Certora Prover proved the implementation of the contract above is correct with respect to formal specifications written by the Certora team. The team also performed a manual audit of the contract.

The formal specifications focus on validating the correct behavior for `YieldBox.sol` as described by the Tapioca / BoringCrypto team and the contract documentation. The rules verify:

- Valid states for the system
- Proper transitions between states
- Method integrity
- High-level properties

These four often describe more than one element of the system and can be cross-system in some instances. The formal specifications have been submitted as a [pull request](#) for BoringCrypto's public git repository.

Previous report

All previously discovered issues can be found in the previous [report](#).

Main Issues Discovered

Severity: Medium

Issue:	Incorrect required permissions in transferMultiple()
Description:	In the function <code>transferMultiple()</code> , the <code>assetId</code> argument which is passed to the modifier is incorrectly set to a max of <code>uint256</code> instead of to the specific <code>assetId</code> . This results in a functional bug, depriving an address with the permissions for that <code>assetId</code> the ability to call that function. This also has the potential to cause a security breach. If one wants to allow someone to act on their behalf for a specific <code>assetId</code> using the <code>transferMultiple()</code> function, it must also allow access for all other assets.
Response:	Fixed in the commit b3c804f4 .

Severity: Low

Issue:	Misleading emit messages in transferMultiple()
Description:	There is a possibility of generating misleading emit messages in the function <code>transferMultiple()</code> . More specifically, it is possible for a user to generate a log message that claims a transfer of an arbitrary large amount from one address to itself. This behavior is unexpected as generally, one should not transfer more funds/shares than they own. The function <code>transferMultiple()</code> first adds the number of shares (received as an argument) to the recipient and only then subtracts it from the sender's balance. This allows a user to transfer any amount of shares to themselves.
Response:	<i>"We accept this behaviour as we prefer to keep on-chain performance over metadata emission."</i>

Informational

Issue:	Registration of two different assets with the same strategy
Description:	YieldBox relies on the strategy to enforce that it will only be registered once (by providing the <code>tokenType</code> , <code>contractAddress</code> , and <code>tokenId</code>). This is an assumption that might not hold practically. As one can imagine, take a scenario in which a previously "malicious" strategy was registered twice with two different <code>tokenTypes</code> . After a code update (and a revocation of the owner's right to further update the strategy), it could function as a legitimate and verified strategy. In this

Issue:	Registration of two different assets with the same strategy
	hypothetical scenario, there is a possibility to achieve a situation in which a strategy that supports one kind of token will be asked by YieldBox to withdraw those tokens in the context of another token (through a different assetId), stealing funds as a result.
Response:	<i>"Asset and strategy within YieldBox by design are supposed to be freely created, and participating to those strategies is at the discretion of the user. As a third party protocol we will only list our own assets to be displayed for users."</i>

Informational

Issue:	Limited amount of possible native tokens
Description:	If an unexpectedly large amount of assets is registered to YieldBox, it would be impossible to create new native tokens. The identifier tokenId is forced to equal the asset's assetId, and then is then casted to a 32 bit long integer. If the assetId exceeds 2^{32} (approximately 4 billion), the casting would revert. Although the number of registered assets is not expected to ever get anywhere near 4 billion, it is still within the range of possibility.
Response:	<i>"We are aware and acknowledge this limitation."</i>

Informational

Issue:	Possibly redundant check that the token's contract exists.
Description:	A comment specifies that the check (AssetRegister, line 56) is necessary for security reasons, e.g. to prevent situations in which one can assume shares of a token that do not yet exist. However, this might be impossible either way as any call to that address before the contract's deployment will revert.
Response:	<i>"We need to verify a 2 case scenario. One where the asset is the native coin, thus having to check for a 0x00 address, and another for non-native tokens, where the address has to be a an existing contract."</i>

Disclaimer

The Certora Prover takes a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. More importantly, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful to you. However, we do not provide warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Summary of formal verification

Notations

✅ Indicates the rule is formally verified on the latest reviewed commit.

❌ Indicates the rule was violated under one of the tested versions of the code.

👉 Indicates the rule is not yet formally specified.

🕒 Indicates that some functions cannot be verified because the rules timed out.

Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

In this document, verification conditions are either shown as logical formulas or Hoare triples of the form $\{p\} C \{q\}$. A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition.

Hoare triples of the form $\{p\} C \{q\}$ hold if any non-reverting execution of program C that starts in a state satisfying the precondition p ends in a state satisfying the postcondition q . The notation $\{p\} C@withrevert \{q\}$ is similar but applies to both reverting and non-reverting executions. Preconditions and postconditions are similar to the Solidity `require` and `assert` statements.

The syntax $\{p\} (C1 \sim C2) \{q\}$ is a generalization of Hoare rules, called relational properties. $\{p\}$ is a requirement on the states before $C1$ and $C2$, and $\{q\}$ describes the states after their executions. Notice that $C1$ and $C2$ result in different states.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts. In some cases however, they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Assumptions and simplifications for verification

- Loops are unrolled. Violations that require a loop to execute more than three times will not be detected.
- Simple strategy mock contract (`SimpleMintStrategy.sol`) was created in order to check all functionalities of YieldBox including communication with strategies.
- For the invariant "sharesToTokensRatio" we assumed simplified computations in `_toShares()` and `_toAmount()` functions in `YieldBoxRebase.sol` . The new ratio is fixed and equal to 2 shares per token.
- Functions `uri()` , `name()` , `symbol()` , `decimals()` were omitted because their implementation significantly slowed down runtime. These omissions do not affect the verification because the functions in question do not impact the contract's state.

Verification of YieldBox.sol

Summary

The contract is an extension of BentoBox (by the Sushi team) that allows an unlimited number of strategies for each token.

Warning Note

The scope of this verification ignores cases of malicious strategies and tokens. Any such malicious component can break properties for assets that are related to the specific component. All users must be aware of this possibility, and avoid using any non-standard token or any strategy that can change its interface with Yieldbox. Such a strategy could be dynamically changing the tokenType etc. For example, double entry ERC20s are ERC20 tokens that have 2 addresses for the same token. In this case, there could be asset duplication. A simple attack might be a user using the 1st address and an attacker registering the 2nd asset in possession of all its shares for it. In this case, the attacker could drain all the funds of the token, effectively draining the users of the 1st asset.

Properties

(✓) `mapArrayCorrealtion` : Mapping ids and array assets are correlated: if assets are equal their indexes must be equal; an asset's fields applied to a mapping should return the same id as the array index.

```
((i < assets.length && j < assets.length)
    => assets[i] == assets[j] <=> i == j))
&& (i < assets.length
    => ids(assets[i].tokenType, assets[i].contractAddress, assets[i].s
&& (j < assets.length
    => ids(assets[j].tokenType, assets[j].contractAddress, assets[j].s
```

(✓) `assetIdtoAssetLength` : Existing asset should have id less than `assets.length` .

```
ids(assets[i].tokenType, assets[i].contractAddress, assets[i].strategy, asse
```

(✓) `erc20HasTokenIdZero` : An asset of type ERC20 must have a tokenId == 0.

```
asset.tokenType == TokenType.ERC20
    && asset.tokenId != 0
=> ids(asset.tokenType, asset.contractAddress, asset.strategy, asset.tokenId
```

(✓) `balanceOfAddressZeroYieldBox` : Balance of address Zero equals Zero.

```
balanceOf(0, tokenId) == 0
```

(✓) `tokenTypeValidity` : The only assetId = 0 should be `TokenType.None` .

```
assets.length > 0
&& (assets[ids(asset)].tokenType == TokenType.None
    <=> ids(asset) == 0)
```

(✓) `sharesSolvency` : Shares solvency: sum of shares of all users should be less than or equal to `totalSupply` of shares.

```
forall tokenId. sumOfShares[tokenId] <= totalSupply[tokenId]
```

(✓) `withdrawIntegrity` : Integrity of withdraw().

```
{
    strategyBalanceBefore = _tokenBalanceOf(asset)
```

```

    && balanceBefore = balanceOf(from, assetId)
  }
  amountOut, shareOut = withdraw(assetId, from, to, amount, share)
  {
    (shareOut == 0 => amountOut == 0)
    && asset.tokenType != TokenType.ERC721
      => (amountOut == 0 && shareOut == 0
        <=> amount == 0 && share == 0)
    && asset.tokenType == TokenType.ERC721
      => (amountOut == 0 && shareOut == 0)
    && (balanceBefore == 0 => shareOut == 0)
    && (strategyBalanceBefore == 0 && asset.strategy != 0
      => amountOut == 0 || shareOut == 0)
  }

```

(❌) `yieldBoxETHAlwaysZero` : `YieldBox` eth balance is unchanged (there is no way to transfer funds to `YieldBox` within the contract's functions).

```

{
  ethBalanceOfAdress(currentContract) == 0
}
< call to any function f >
{
  ethBalanceOfAdress(currentContract) == 0
}

```

- Bug " `DepositETHAsset()` " uses a different amount than provided `msg.value` ".

(✅) `strategyCorrelatesAsset` : If an asset has a strategy, it should have the same fields as that strategy.

```

asset.strategy == Strategy
=> (asset.tokenType == Strategy.tokenType()
    && asset.contractAddress == Strategy.contractAddress()
    && asset.tokenId == Strategy.tokenId())

```

(✅) `transferIntegrity` : Correctness of `transfer()` function:

- `from` and `to` balances are updated correctly
- another user's balance remains untouched
- `totalSupply` doesn't change

```

{
  balanceFromBefore = balanceOf(from, assetId)
  && balanceToBefore = balanceOf(to, assetId)
  && balanceRandBefore = balanceOf(rand, assetId)

```

```

    && totalSupplyBefore = totalSupply(assetId)
}
transfer(from, to, assetId, share)
{
    balanceFromAfter = balanceOf(from, assetId)
    && balanceToAfter = balanceOf(to, assetId)
    && balanceRandAfter = balanceOf(rand, assetId)
    && totalSupplyAfter = totalSupply(assetId)
    && balanceFromBefore - balanceFromAfter == balanceToAfter - balanceToB
        && (from != to => balanceFromBefore - balanceFromAfter == share)
    && (rand != from && rand != to) => balanceRandBefore == balanceRandAft
    && totalSupplyBefore == totalSupplyAfter
}

```

- the same property was implemented for `batchTransfer()` and `transferMultiple()`.

(✓) `transferIntegrityRevert` : Transfer should revert if `msg.sender` is not allowed.

```

{
    allApprovalBefore = isApprovedForAll(from, msg.sender)
    && assetsApprovalBefore = isApprovedForAsset(from, msg.sender, assetId)
}
transfer@withrevert(from, to, assetId, share)
{
    (e.msg.sender != from
        && !allApprovalBefore
        && !assetsApprovalBefore)
    => transferReverted
}

```

(✓) `permitShouldAllow` : Correctness of permit functions: approval should be granted.


```

{
    allApprovalBefore = isApprovedForAll(from, msg.sender)
    && assetsApprovalBefore = isApprovedForAsset(from, msg.sender, assetId)
}
< call permit() or permitAll() >
{
    allApprovalAfter = isApprovedForAll(owner, spender)
    && assetsApprovalAfter = isApprovedForAsset(owner, spender, assetId)
    && !allApprovalBefore && !assetsApprovalBefore
        => allApprovalAfter || assetsApprovalAfter
    && !allApprovalBefore && allApprovalAfter
        => permitAll() was called
    && !assetsApprovalBefore && assetsApprovalAfter
}


```




```
    => permit() was called
}
```

 **correctSharesWithdraw** : The correct amount of shares will be withdrawn depending on the token type.

```
{
    sharesBefore = balanceOf(from, assetId)
    && sharesToWithdraw = toShare(assetId, amount, true)
}
amountOut, shareOut = withdraw(assetId, from, to, amount, share)
{
    sharesAfter = balanceOf(from, assetId)
    && getAssetTokenType(assetId) == TokenType.ERC721
    => sharesBefore - sharesAfter == 1
    && getAssetTokenType(assetId) != TokenType.ERC721 && amount == 0
    => sharesBefore - sharesAfter == share
    getAssetTokenType(assetId) != TokenType.ERC721 && share == 0
    => (sharesBefore - sharesAfter == sharesToWithdraw
        && sharesToWithdraw == shareOut)
}
```


 **withdrawForNFTReverts** : Catches the bug "Withdraw cannot succeed for ERC721 assets".

```
{ }
amountOut, shareOut = withdraw@withrevert(assetId, from, to, amount, sha
{
    lastReverted => (TokenType != TokenType.ERC20
        || TokenType != TokenType.ERC721
        || TokenType != TokenType.ERC1155)
}
```

 **tokenInterfaceConfusion** : Catches the bug "Interface "confusion" between ERC721 and ERC20...".


```
{
    erc20BalanceBefore = ERC20.balanceOf(randomAddress)
}
depositNFTAsset(assetId, from, to)
{
    erc20BalanceAfter = ERC20.balanceOf(randomAddress)
    && ((asset.tokenType == YieldData.TokenType.ERC721
        && asset.contractAddress == dummyERC20)
```

```
    => erc20BalanceBefore == erc20BalanceAfter)
}
```


() `sharesAfterDeposit` : Catches the bug "First depositor can steal value of subsequent deposits".

```
{ }
    amountOut, shareOut = deposit(tokenType, contractAddress, strategy, tokenType, amount)
{
    (amount > 0 && minShareOut > 0) => shareOut > 0
}
```

- if `minShareOut` was set incorrectly there is still a possibility to lose shares.

() `depositETHCorrectness` : Catches the bug "DepositETHAsset() uses a different amount than provided msg.value".

```
{
    balanceBefore = wrappedNative.balance
}
    amountOut, shareOut = depositETHAsset(assetId, to, amount)
{
    balanceAfter = wrappedNative.balance
    && balanceAfter == balanceBefore + msg.value
}
```

() `dontBurnSharesWithdraw` : Catch the bug from previous report "ERC721 assets can be withdrawn without burning any shares".

```
{
    ownerBefore = ERC721.ownerOf(asset.tokenId)
    && sharesBefore = balanceOf(from, assetId)
}
    amountOut, shareOut = withdraw(assetId, from, to, amount, share)
{
    ownerAfter = ERC721.ownerOf(asset.tokenId)
    && sharesAfter = balanceOf(from, assetId)
    && sharesBefore == sharesAfter => ownerBefore == ownerAfter
}
```