



# Formal Verification of YieldBox (Tapioca)

---

## 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 [e8c74911](#).

The scope of this verification is Tapioca / BoringCrypto's contract `YieldBox.sol`.

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 these contracts.

The formal specifications focus on validating 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, and high-level properties (which often describe more than one element of the system and can even be cross-system).

The formal specifications have been submitted as a [pull request](#) for BoringCrypto's public git repository.

## Main Issues Discovered

---

Severity: High

Issue:	ERC721 assets can be withdrawn without burning any shares
Description:	The <code>withdraw()</code> function allows a user to withdraw a given NFT even if they provide 0 shares in exchange. This effectively enables any NFT in Yieldbox to be grabbed by anyone.

Issue:	<b>ERC721 assets can be withdrawn without burning any shares</b>
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: <i>"Withdraw can simply check that amount == 1 for ERC721 assets. Adding ERC721 was one of the last changes and I'm still not sure about it. I think having a TokenType of ERC721 pool could also be useful."</i>

Severity: High

Issue:	<b>First depositor can steal value of some subsequent deposits</b>
Description:	<p>Because the first depositor has all the shares by default, they can transfer a large amount directly to the contract and distort the ratio between <code>totalAmount</code> (tokens in yieldbox/strategy) and <code>totalShares</code>, resulting in 0 shares in the <code>toShares()</code> calculation (or an unexpectedly large amount in the <code>toAmount()</code> function).</p> <p>The attacker doesn't lose any of their funds doing this, so there's no risk for them using a large amount. Furthermore, an attacker can make the large transfer via a front-run, to undermine unsuspecting users that try to check the system's state before sending their deposit.</p>
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: <i>"The solution to this and related issues is to not have NO_STRATEGY assets at all (except for Native tokens of course, but they can't be deposited or withdrawn as they are native. They can of course be deposited or withdrawn as ERC1155 tokens, in which case they will be a new asset with a strategy). This way the YieldBox contract will never hold any tokens. If a first depositor warps the ratio, the UI can simply ignore this and assume this asset hasn't been created yet and create a new strategy for it (even for "blank" strategies), leaving the attacker with a loss and an unused pointless strategy contract. Providing the UI filters out compromised assets."</i>

Severity: High

Issue:	<b>Interface "confusion" between ERC721 and ERC20 enables two distinct assets to influence each other's <code>_tokenBalanceOf()</code></b>
Description:	When depositing or withdrawing an asset, either <code>TransferFrom()</code> or <code>safeTransferFrom()</code> of the token is called. Since those have the same signature for both ERC721 and ERC20, it's possible to deposit two distinct assets with identical <code>tokenAddress</code> but with different

Issue:	<b>Interface "confusion" between ERC721 and ERC20 enables two distinct assets to influence each other's <code>_tokenBalanceOf()</code></b>
	<p><code>tokenType</code> . This breaks Yieldbox's premise of independent assets.</p> <p>Consider the following scenario:          If the actual token is an ERC20, it would also possible to deposit via the NFT / ERC720 <code>assetId</code> (in which case <code>tokenId</code> would become the amount transferred in). So long as it remains possible to withdraw this NFT asset (which depends on the fix to the current version), the attacker will be able to drain the ERC20 asset.</p> <p>To do that, the attacker also has to have shares in the ERC20 asset. When they deposit with the NFT asset, the <code>tokenBalanceOf()</code> for the ERC20 asset goes up (and so does the value of their shares). Then the attacker withdraws from the ERC20 asset, and withdraws from the NFT asset - retaining the value of the deposit to the 2nd asset but having a gain on the 1st asset (at the expense of the other users).</p>
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: reasoning is the same as in the issue above.

**Severity: Medium**

Issue:	<b>First depositor can block subsequent deposits from other users (DoS)</b>
Description:	<p>This is caused by a new argument introduced to some functions in order to fix a bug "First depositor can steal value of some subsequent deposits." The scenario is almost the same, but in this case, the second depositor won't be able to deposit because <code>shareOut</code> won't satisfy the <code>minShareOut</code> requirement, provided that <code>minShareOut</code> was calculated correctly.</p> <p>The attacker doesn't lose any of their funds doing this, so there's no risk for them using a large amount. Furthermore, an attacker can make the large transfer via a front-run, to undermine unsuspecting users who try to check the system's state before sending their deposit.</p>
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: reasoning is the same as in the issue above.

## Severity: Medium

Issue:	<b>Withdraw cannot succeed for ERC721 assets (that don't have strategy)</b>
Description:	This is caused by a wrong <code>tokenType</code> check inside the call to <code>_tokenBalanceOf()</code> , which will call the IERC1155 on the NFT asset. Since this interface is different than the underlying token, it will always revert.
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: <i>"This was the result of the code not actually being finished yet. With the change to always use strategies, this issue has gone away."</i>

## Severity: Low

Issue:	<b>DepositETHAsset() uses a different amount than provided msg.value</b>
Description:	<p>When depositing ETH, the system converts it to wrappedETH and stores it as an ERC20 asset. The amount of wrappedETH created and allocated is derived only from the user input amount, regardless of how much was paid via <code>msg.value</code>.</p> <p>This can lead to a leftover balance in Yieldbox (which could be considered a mistake on the part of the user), or a situation where any user could claim the existing ETH balance of Yieldbox.</p> <p>Since it seems that YieldBox shouldn't really ever have any such balance from "normal" activity, this seems to qualify as a low-severity.</p>
Response:	The issue was fixed by Tapioca team. However, further changes will be made to match BoringCrypto resolutions: <i>"This is by design. Changing this to msg.value is still safe in BoringBatchable, but it breaks sending for example sending 1 ETH to YieldBox and using a batch to put 0.3 ETH in one asset and 0.7 in another. Considering gas costs I don't believe it's the job of protocol contracts to provide safety checks for bad UIs. If this is a concern and gas usage is not, you could write a wrapper contract with more safety checks."</i>

## Informational

Issue:	<b>BoringBatchable's Batch() function can delegate-call a payable function multiple times</b>
Description:	<p>A delegate call perserves the environment <code>e</code> , which means it could call mutiple times with the same <code>msg.value</code> .</p> <p>Using a similar example as above, it would be possible for anyone to claim the ETH balance of YieldBox (even if it was impossible with a single external call, it would be possible via <code>batch()</code> ).</p> <p>Since it seems that YieldBox shouldn't really ever have any such balance from "normal" activity, this seems to qualify as a low-severity. We highlight this issue here so as to recommend special care be taken with any features regarding <code>msg.value</code> or payable functions (e.g. on alterations or new features).</p>
Response:	<p>The issue was fixed by Tapioca team. However, futher changes will be made to match BoringCrypto resolutions: <i>"This is a quite well known issue now. I've put a big warning for this in BoringBatchable. YieldBox doesn't use msg.value, so this isn't an issue."</i></p>

## Disclaimer


The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. 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, but provide no 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 ~ 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, but in some cases 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

---

- We unroll loops. Violations that require a loop to execute more than once 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 malicious tokens. Any such malicious component can break properties for assets that are related to that component. All users should be aware of this possibility, and avoid using any non-standard token, or any strategy that can change its interface with Yieldbox (like dynamically changing the tokenType etc). For example - double entry ERC20s are ERC20 tokens that have 2 addresses for the same token. In such a case there could be asset duplication. A simple attack might be a user using the 1st address and an attacker registering the 2nd asset and having all the shares for it. In that case, the attacker could drain all the funds of that 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
```

(✓) `balanceOfAddressZero` : 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)
```

(✓) `sharesToTokensRatio` : Checking a solvency property regarding the ratio between shares and amount of tokens for non-NFT functions: total shares <= total amount of tokens \* 10<sup>8</sup>. Except NFT related functions: `withdrawNFT()`, `depositNFT()`, `depositNFTAsset()`.

```
assetId > 0 => (totalSupply(assetId) <= _tokenBalanceOf(asset) * 2)
```

- Was simplified to 2:1 ratio because of difficulty math operations.
- `assetId == 0` is not checked because it's created by the constructor as a token that shouldn't be operated to keep correlation between `ids` and `assets` where id 0 should mean that an asset is not registered yet.

(✓) `sharesToTokensRatioNFT` : Checking solvency property regarding ratio between shares and amount of tokens for NFT: total shares == total amount of NFTs.

```
totalSupply(assetId) <= _tokenBalanceOf(asset)
```

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

```
{  
    strategyBalanceBefore = _tokenBalanceOf(asset)  
    && balanceBefore = balanceOf(from, assetId)  
}  
amountOut, shareOut = withdraw(assetId, from, to, amount, share)  
{  
    (shareOut == 0 => amountOut == 0)  
    && (amountOut == 0 && shareOut == 0  
        <=> amount == 0 && share == 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(e, currentContract) == 0
}
< call to any function f >
{
  ethBalanceOfAdress(e, currentContract) == 0
}

```

(✓) **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())

```

## Bug catching properties

(✓) **nftWithdrawReverts** : Catches the bug "Withdraw cannot succeed for ERC721 assets".

```

{ }
amountOut, shareOut = withdraw@withrevert(assetId, from, to, amount, sha
{
  !lastReverted => (assets[assetId].tokenType == TokenType.ERC721
    || assets[assetId].strategy == 0
    || assets[assetId].contractAddress == dummyERC721)
}

```

- Implementation was updated after a fix release:

```

{ }
amountOut, shareOut = withdrawNFT@withrevert(assetId, from, to)
{
  !lastReverted => (assets[assetId].tokenType == TokenType.ERC721
    || assets[assetId].strategy == 0
    || assets[assetId].contractAddress == dummyERC721)
}

```

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

```

{
    asset.tokenType == TokenType.ERC721
    && asset.contractAddress == address(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 some subsequent deposits".

```

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

```

- there is a possibility of DoS as explained in "Main Issues Discovered" section: "First depositor can block subsequent deposits from other users (DoS)".
- 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
}

```

(✓) `dontBurnShares` : Catch the bug "ERC721 assets can be withdrawn without burning any shares".

```

{
    amount == 0 && share == 0
    && balanceBefore = balanceOf(from, assetId)
}

```

```
withdraw@withrevert(e, assetId, from, to, amount, share)
{
    balanceBefore == 0 => lastReverted
}
```

- Implementation was updated after a fix release:

```
{
    amount == 0 && share == 0
    && balanceBefore = balanceOf(from, assetId)
}
withdrawNFT@withrevert(e, assetId, from, to)
{
    balanceBefore == 0 => lastReverted
}
```