BadgerDAO ibBTC Wrapper contest Findings & Analysis Report

2021-12-16

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of the BadgerDAO ibBTC Wrapper smart contract system written in Solidity. The code contest took place between October 28—October 30 2021.

Wardens

20 Wardens contributed reports to the BadgerDAO ibBTC Wrapper contest:

1. hyh
2. WatchPug (jtp and ming)
3. gzeon
4. jonah1005
5. kenzo
6. cmichel
7. defsec

8. hack3r-0m
9. JMukesh
10. pauliax
11. pants
12. TomFrench
13. pmerkleplant
14. loop
15. gpersoon
16. leastwood
17. jah
18. ye0lde
19. ych18

This contest was judged by leastwood.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 23 unique vulnerabilities and 51 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 15 received a risk rating in the category of LOW severity.

C4 analysis also identified 12 non-critical recommendations and 16 gas optimizations.

Scope

The code under review can be found within the C4 BadgerDAO ibBTC Wrapper contest repository, and is composed of 90 smart contracts written in the Solidity programming language and includes 7479 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] The design of `wibBTC` is not fully compatible with the current Curve StableSwap pool

Submitted by WatchPug, also found by gzeon

Per the documentation, `wibBTC` is designed for a Curve StableSwap pool. However, the design of `wibBTC` makes the balances change dynamically and automatically. This is unusual for an ERC20 token, and it's not fully compatible with the current Curve StableSwap pool.

Specifically, a Curve StableSwap pool will maintain the balances of its `coins` based on the amount of tokens added, removed, and exchanged each time. In another word, it can not adopt the dynamic changes of the balances that happened automatically.

The pool's actual dynamic balance of `wibBTC` will deviate from the recorded balance in the pool contract as the `pricePerShare` increases.

Furthermore, there is no such way in Curve StableSwap similar to the `sync()` function of UNI v2, which will force sync the stored `reserves` to match the balances.

PoC

Given:

- The current `pricePerShare` is: `1`;
- The Curve pool is newly created with 0 liquidity;
- Alice added `100 wibBTC` and `100 wBTC` to the Curve pool; Alice holds 100% of the pool;
- After 1 month with no activity (no other users, no trading), and the `pricePerShare` of `ibBTC` increases to `1.2`;
- Alice removes all the liquidity from the Curve pool.

While it's expected to receive `150 wibBTC` and `100 wBTC`, Alice actually can only receive `100 wibBTC` and `100 wBTC`.

Recommended Mitigation Steps

Consider creating a revised version of the Curve StableSwap contract that can handle dynamic balances properly.

dapp-whisperer (BadgerDAO) confirmed:

> We will be creating a custom pool that takes this into account based on the rate_multiplier variable on the MetaPools.

- Draft implementation

[H-02] Approved spender can spend too many tokens

Submitted by cmichel, also found by WatchPug, jonah1005, gzeon, and TomFrench

The `approve` function has not been overridden and therefore uses the internal *shares*, whereas `transfer(From)` uses the rebalanced amount.

Impact

The approved spender may spend more tokens than desired. In fact, the approved amount that can be transferred keeps growing with `pricePerShare`.

Many contracts also use the same amount for the `approve` call as for the amount they want to have transferred in a subsequent `transferFrom` call, and in this case, they approve an amount that is too large (as the approved `shares` amount yields a higher rebalanced amount).

Recommended Mitigation Steps

The `_allowances` field should track the rebalanced amounts such that the approval value does not grow. (This does not actually require overriding the `approve` function.) In `transferFrom`, the approvals should then be subtracted by the *transferred* `amount`, not the `amountInShares`:

```
// _allowances are in rebalanced amounts such that they don't gr
// need to subtract the transferred amount
_approve(sender, _msgSender(), _allowances[sender][_msgSender()]
```

tabshaikh (Badger) confirmed and resolved:

> Fix here: https://github.com/Badger-Finance/rebasing-ibbtc/pull/7

[H-03] `WrappedIbbtcEth` contract will use stalled price for mint/burn if `updatePricePerShare` wasn't run properly

Submitted by hyh, also found by cmichel, gpersoon, leastwood, hack3r-0m, kenzo, WatchPug, and loop

Impact

Malicious user can monitor `SetPricePerShare` event and, if it was run long enough time ago and market moved, but, since there were no `SetPricePerShare` fired, the contract's `pricePerShare` is outdated, so a user can `mint()` with `pricePerShare` that is current for contract, but outdated for market, then wait for price update and `burn()` with updated `pricePerShare`, yielding risk-free profit at expense of contract holdings.

Proof of Concept

`WrappedIbbtcEth` updates `pricePerShare` variable by externally run `updatePricePerShare` function. The variable is then used in mint/burn/transfer functions without any additional checks, even if outdated/stalled. This can happen if the external function wasn't run for any reason. The variable is used via `balanceToShares` function: `WrappedIbbtcEth.sol` L155

This is feasible as `updatePricePerShare` to be run by off-chain script being a part of the system, and malfunction of this script leads to contract exposure by stalling the price. The malfunction can happen both by internal reasons (bugs) and by external ones (any system-level dependencies, network outrages).
`updatePricePerShare` function: `WrappedIbbtcEth.sol` L72

Recommended Mitigation Steps

The risk comes with system design. Wrapping price updates with contract level variable for gas costs minimization is a viable approach, but it needs to be paired with corner cases handling. One of the ways to reduce the risk is as follows:

Introduce a threshold variable for maximum time elapsed since last `pricePerShare` update to `WrappedIbbtcEth` contract.

Then 2 variants of `transferFrom` and `transfer` functions can be introduced, both check condition {now - time since last price update < threshold}. If condition holds both variants, do the transfer. If it doesn't, then the first variant reverts, while the second do costly price update. I.e. it will be cheap transfer (that works only if price is recent) and full transfer (that is similar to the first when price is recent, but do price update on its own when price is stalled). This way, this full transfer is guaranteed to run and is usually cheap, costing more if price is stalled and it does the update.

After this, whenever scheduled price update malfunctions (for example because of network conditions), the risk will be limited by market volatility during threshold time at maximum, i.e. capped.

See issue page for example code:

dapp-whisperer (BadgerDAO) confirmed:

> Agreed, appreciate the thorough breakdown. We will add a "max staleness" to the ppfs update.

> I do see some merit in the idea of "updating when needed" at expense of the next user, but due to interface considerations we'd like to keep that consistent for users. In practice, we will run a bot to ensure timely updates.

> The pps updates are small and infrequent.

[H-04] `WrappedIbbtc` and `WrappedIbbtcEth` contracts do not filter out price feed outliers

Submitted by hyh

Impact

If price feed is manipulated in any way or there is any malfunction based volatility on the market, both contracts will pass it on a user. In the same time it's possible to construct mitigation mechanics for such cases, so user economics would be affected by sustainable price movements only. As price outrages provide a substantial attack surface for the project it's worth adding some complexity to the implementation.

Proof of Concept

In `WrappedIbbtcEth pricePerShare` variable is updated by externally run `updatePricePerShare` function (WrappedIbbtcEth.sol L72), and then used in mint/burn/transfer functions without additional checks via `balanceToShares` function: `WrappedIbbtcEth.sol` L155

In `WrappedIbbtc` price is requested via `pricePerShare` function(`WrappedIbbtc.sol` L123), and used in the same way without additional checks via `balanceToShares` function.

Recommended Mitigation Steps

Introduce a minting/burning query that runs on a schedule, separating user funds contribution and actual mint/burn. With user deposit or burn, the corresponding action to be added to commitment query, which execution for mint or redeem will later be sparked by off-chain script according to fixed schedule. This also can be open to public execution with gas compensation incentive, for example as it's done in Tracer protocol: `PoolKeeper.sol` L131

Full code of an implementation is too big to include in the report, but viable versions are available publicly (Tracer protocol version can be found at the same repo, `implementation/PoolCommitter` sol).

Once the scheduled mint/redeem query is added, the additional logic to control for price outliers will become possible there, as in this case mint/redeem execution can be conditioned to happen on calm market only, where various definitions of calm can be implemented. One of the approaches is to keep track of recent prices and require that new price each time be within a threshold from median of their array.

Example:

```
// Introduce small price tracking arrays:
uint256\[] private times;
uint256\[] private prices;

// Current position in array
uint8 curPos;

// Current length, grows from 0 to totalMaxPos as prices are bei
uint8 curMaxPos;

// Maximum length, we track up to totalMaxPos prices
uint8 totalMaxPos = 10;

// Price movement threshold
uint256 moveThreshold = 0.1\*1e18;
```

We omit the full implementation here as it is lengthy enough and can vary. The key steps are:

- Run query for scheduled mint/redeem with logic: if next price is greater than median of currently recorded prices by threshold, add it to the records, but do not mint/redeem.
- That is, when scheduled mint/redeem is run, on new price request, WrappedIbbtcEth.core.pricePerShare() or WrappedIbbtc.oracle.pricePerShare(), get newPrice and calculate current price array median, curMed
- prices[curPos] = newPrice
- if (curMaxPos < totalMaxPos) {curMaxPos += 1}
- if (curPos == curMaxPos) {curPos = 0} else {curPos += 1}
- if (absolute*value*of(newPrice - curMed) < moveThreshold * curMed / 1e18) {do*mint/redeem; return* \0_status}
- else {return_1_status}

Schedule should be frequent enough, say once per 30 minutes, which is kept while returned status is 0. While threshold condition isn't met and returned status is 1, it runs once per 10 minutes. The parameters here are subject to calibration.

This way if the price movement is sustained the mint/redeem happens after price array median comes to a new equilibrium. If price reverts, the outbreak will not have material effect mint/burn operations. This way the contract vulnerability is considerably reduced as attacker would need to keep distorted price for period long enough, which will happen after the first part of deposit/withdraw cycle. I.e. deposit and mint, burn and redeem operations will happen not simultaneously, preventing flash loans to be used to elevate the quantities, and for price to be effectively distorted it would be needed to keep it so for substantial amount of time.

dapp-whisperer (BadgerDAO) confirmed:

> Minting and burning happens atomically within larger function calls and our current approach isn't amenable to this change.

Medium Risk Findings (4)

[M-01] Unable to transfer `WrappedIbbtc` if Oracle go down

Submitted by gzeon

Impact

In `WrappedIbbtc`, user will not be able to transfer if `oracle.pricePerShare()` (L124) revert. This is because `balanceToShares()` is called in both transfer and `transferFrom`, which included a call to `pricePerShare()`.

If this is the expected behavior, note that `WrappedIbbtcEth` is behaving the opposite as it uses the cached value in a local variable `pricePerShare`, which is only updated upon call to `updatePricePerShare()`.

Recommended Mitigation Steps

Depending on the specification, one of them need to be changed.

dapp-whisperer (BadgerDAO) confirmed

[M-02] Null check in `pricePerShare`

Submitted by hack3r-0m, also found by defsec

oracle can `0` as a price of the share, in that case, 0 will be the denominator in some calculations which can cause reverts from SafeMath (for e.g here: `WrappedIbbtc.sol` **L148**) resulting in Denial Of Service.

- `WrappedIbbtcEth.sol` **L73**
- `WrappedIbbtc.sol` **L123**

Recommended Mitigation Steps

Add a null check to ensure that on every update, the price is greater than 0.

dapp-whisperer (BadgerDAO) confirmed:

> Agreed. we will implicitly or explicitly add this check.

[M-03] hard to clear balance

Submitted by jonah1005

The contract does not allow users to transfer by share. Therefore, It is hard for users to clear out all the shares. There will be users using this token with Metamask and it is likely the `pricePerShare` would increase after the user sends transactions. I consider this is a medium-risk issue.

Proof of Concept

WrappedIbbtc.sol#L110-L118

Recommended Mitigation Steps

A new `transferShares` beside the original `transfer()` would build a better UX. sushi's bento box would be a good ref BentoBox.sol

dapp-whisperer (BadgerDAO) confirmed

[M-04] No sanity check on `pricePerShare` might lead to lost value

Submitted by kenzo

`pricePerShare` is read either from an oracle or from ibBTC's core.

If one of these is bugged or exploited, there are no safety checks to prevent loss of funds.

Impact

As `pricePerShare` is used to calculate transfer amount, a bug or wrong data retuning a smaller `pricePerShare` than it really is, could result in drainage of wibbtc from Curve pool.

Proof of Concept

Curve's swap and remove liquidity functions will both call wibbtc's `transfer` function:

- https://etherscan.io/address/0xFbdCA68601f835b27790D98bbb8eC7f05FDEaA9B#code%23L790
- https://etherscan.io/address/0xFbdCA68601f835b27790D98bbb8eC7f05FDEaA9B#code%23L831
- The `transfer` function calculates the amount to send by calling `balanceToShares`: `WrappedIbbtcEth.sol` L127
- `balanceToShares` calculates the shares (=amount to send) by dividing in `pricePerShare`: `WrappedIbbtcEth.sol` L156

Therefore, if due to a bug or exploit in ibBTC core / the trusted oracle `pricePerShare` is smaller than it really is, the amount that will be sent will grow larger. So Curve will send to the user/exploiter doing swap/remove liquidity more tokens that he deserves.

Tools Used

Manual analysis, hardhat

Recommended Mitigation Steps

Add sanity check:

`pricePerShare` should never decrease but only increase with time (as ibbtc accrues interest) (validated with DefiDollar team). This means that on every `pricePerShare` read/update, if the new `pricePerShare` is smaller than the current one, we can discard the update as bad data.

This will prevent an exploiter from draining Curve pool's wibbtc reserves by decreasing `pricePerShare`.

dapp-whisperer (BadgerDAO) confirmed

Low Risk Findings (15)

- [L-01] The `value` parameter of the `Transfer` event is wrong *Submitted by WatchPug*
- [L-02] pendingGovernance and Governace address can be same *Submitted by JMukesh, also found by hack3r-0m*
- [L-03] use of depreciated “now” *Submitted by JMukesh*
- [L-04] In `updatePricePerShare()` no value is returned *Submitted by JMukesh, also found by pauliax*
- [L-05] `updatePricePerShare` should be run atomically with `setCore()` to make sure `pricePerShare` is up-to-date with the new Core *Submitted by WatchPug*
- [L-06] Redundant use of `virtual` *Submitted by WatchPug*
- [L-07] `initialize` functions can be frontrun *Submitted by cmichel, also found by pants, ych18, and defsec*
- [L-08] Missing parameter validation *Submitted by cmichel, also found by jah, pants, and JMukesh*
- [L-09] Pending governance is not cleared *Submitted by cmichel*
- [L-10] Add zero address validation in the setPendingGovernance function *Submitted by defsec, also found by JMukesh*
- [L-11] Deprecated Function Usage *Submitted by defsec*
- [L-12] PREVENT DIV BY 0 *Submitted by defsec, also found by pauliax*
- [L-13] use safeTransfer instead of transfer of ibbtc *Submitted by pants*
- [L-14] Consider making contracts Pausable *Submitted by pauliax*
- [L-15] Lack of `address(0)` check *Submitted by pmerkleplant*

Non-Critical Findings (12)

- [N-01] use of floating pragma *Submitted by JMukesh, also found by loop and pants*
- [N-02] Outdated compiler version *Submitted by WatchPug*
- [N-03] Outdated versions of OpenZeppelin library *Submitted by WatchPug*
- [N-04] Missing error messages in require statements *Submitted by WatchPug*

- [N-05] Inconsistent use of `_msgSender()` *Submitted by WatchPug, also found by ye0lde and hack3r-0m*
- [N-06] Consider removing `ICore.sol` *Submitted by WatchPug*
- [N-07] Constants are not explicitly declared *Submitted by WatchPug*
- [N-08] Critical changes should use two-step procedure *Submitted by WatchPug*
- [N-09] modified _balances in OZ contract *Submitted by pauliax*
- [N-10] Remove unused functions in dependencys *Submitted by pmerkleplant*
- [N-11] `WrappedIbbtc.sol` implements, but does not inherit, the `ICoreOracle` interface *Submitted by pmerkleplant*
- [N-12] No Initial Ownership Event (WrappedIbbtcEth.sol, WrappedIbbtcEth.sol) *Submitted by ye0lde*

Gas Optimizations (16)

- [G-01] Avoid unnecessary storage read can save gas *Submitted by WatchPug, also found by pants*
- [G-02] Cache external call result in the stack can save gas *Submitted by WatchPug*
- [G-03] Consider caching `pricePerShare` for `WrappedIbbtc.sol` to save gas *Submitted by WatchPug*
- [G-04] Avoid unnecessary external calls and storage writes can save gas *Submitted by WatchPug*
- [G-05] Upgrade pragma to at least 0.8.4 *Submitted by defsec, also found by cmichel*
- [G-06] Gas: Event parameters read from storage *Submitted by cmichel*
- [G-07] Gas Optimization: Retrieve internal variables directly *Submitted by gzeon*
- [G-08] Use Minimal Interface for gas optimizations *Submitted by hack3r-0m*
- [G-09] Immutable variable *Submitted by pauliax, also found by pants and hack3r-0m*
- [G-10] Gas Saving by changing the visibility of initialize function from public to external *Submitted by jah, also found by WatchPug and loop*

- [G-11] missing zero-address check *Submitted by jah*
- [G-12] Events are emitting storage vars instead of user/system values *Submitted by kenzo*
- [G-13] ICore import *Submitted by pauliax*
- [G-14] onlyOracle never used *Submitted by pauliax, also found by WatchPug*
- [G-15] Check if amount is not zero *Submitted by pauliax*
- [G-16] Use existing memory value of state variable (setPendingGovernance) *Submitted by ye0lde*

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top