Blockswap Formal Verification Contest with Certora Findings & Analysis Report

2023-05-04

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 formal verification contest is an event in partnership with Certora Inc. in which community participants, referred to as Wardens, use formal verification tools to mathematically prove the correctness of a program or system in exhange for a bounty provided by sponsoring projects.

During the formal verification contest outlined in this document, Wardens utilized the Certora Prover (verification tool) to conduct formal verification of the Blockswap smart contract system. The formal verification contest took place between January 19—February 02 2023.

Wardens

23 Wardens contributed reports to the Blockswap Formal Verification contest:

1. Koolex
2. neumo
3. jessicapointing
4. eighty
5. Dravee
6. Apocalypto (cRat1st0s, reassor, and M0ndoHEHE)
7. Saintcode_
8. abhi512
9. PPrieditis
10. Quiark
11. horsefacts
12. zapaz
13. Junnon
14. slvDev
15. gzeon

16. 100proof
17. carrotsmuggler
18. stent
19. BAHOZ
20. coqlover
21. Zarf

This contest was judged by Certora Inc.

Final report assembled by liveactionllama.

Summary

Note: during a C4 formal verification contest, Wardens can uncover both “injected bugs” and “real bugs”. This report will focus on the latter.

The C4 formal verification contest yielded an aggregated total of 6 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 6 received a risk rating in the category of MEDIUM severity.

All of the issues are presented below.

Scope

The code under review can be found linked within the C4 Blockswap Formal Verification contest repository, and is composed of 1 smart contract written in the Solidity programming language and includes 402 lines of Solidity code.

Severity Criteria

Teryanarmen from Certora Inc. judged this C4 formal verification contest, and assessed severity of disclosed vulnerabilities based on two primary risk categories: high and medium.

For more information regarding the severity criteria for these two categories, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (6)

[M-01]

Submitted by Dravee

Impact

DOS when using `addPriorityStakers` with two consecutive addresses badly ordered `(address a2 < address a1)`, which can be quite frequent (50% chance).

Proof of Concept

There's the following condition in the `_addPriorityStakers` function at L626:

File: Syndicate.sol

```
623:         for (uint256 i; i < numOfStakers; ++i) {
624:             address staker = _priorityStakers[i];
625:
626:             if (i > 0 && staker < _priorityStakers[i-1]) re
627:
628:             isPriorityStaker[staker] = true;
629:
630:             emit PriorityStakerRegistered(staker);
631:         }
```

As we can see here, after the index 0, it will revert if the address at index i is less than the address at index i - 1, which is quite an odd condition. Additionally, the custom error is `DuplicateArrayElements`, which doesn't match with what the written condition is checking.

When adding, as an example, any 2 addresses as Priority Stakers, whether one address is computed to be greater or less than the previous one shouldn't matter (any permutation of those 2 addresses should enable these 2 addresses to be added as Priority Stakers). We can guess here that the condition was badly implemented, making so that adding a list of Priority Stakers has a 50% chance of failing.

You can try the following on Remix by inputting 2 random addresses and see that this can be true or false depending on the order, hence the 50% chance of failure claim:

```
        function addrCompare(address a1, address a2) external pure r
            return a1 < a2;
        }
```

The following rule catches it as it's unreachable with the bug (original code), and passes without it (suggested remediation):

```
    rule addingTwoDifferentPriorityStackers(address _priorityStaker1
        // Excluding address(0)
        require(_priorityStaker1 != 0 && _priorityStaker2 != 0);
        // Avoiding duplicates and making sure the address at index
        require(_priorityStaker1 > _priorityStaker2);
        // Making sure they aren't already Priority Stakers
        require(!isPriorityStaker(_priorityStaker1) && !isPrioritySt
        env e;

        // Adding any 2 Priority stakers address
        addPriorityStakers(e, _priorityStaker1, _priorityStaker2);
        // The rule will fail due to this assertion being unreachabl
        assert(true, "This is unreacheable");
    }
```

Recommended Mitigation Steps

My guess is that here, the developer wanted to somehow report that the address list contains a mistake.

However, here, even if there were duplicates in the array, this wouldn't change anything regarding the final state (just some gas would be wasted with multiple SSTOREs). I'd advise against checking if the value is already set in storage before writing to it, as multiple SLOADs can make the function call quite gas heavy very fast.

The real condition was probably intended to be "if the staker's index isn't equal to the current index then revert", but for that you'd need a way to fetch an index in an array (like JavaScript's indexOf), which isn't the case in Solidity.

The simplest and best solution here is simply to remove the line:

```
File: Syndicate.sol
        for (uint256 i; i < numOfStakers; ++i) {
            address staker = _priorityStakers[i];

-             if (i > 0 && staker < _priorityStakers[i-1])

            isPriorityStaker[staker] = true;

            emit PriorityStakerRegistered(staker);
        }
```

Again, there's no impact besides wasting gas in adding a Priority Staker multiple times, so this revert shouldn't exist in my opinion

vince0656 (Blockswap) commented:

> Assessment: Low/Medium

> We will instead check `isPriorityStaker[staker]` and revert if true - thanks.

Dravee (warden) commented:

> Hey there @vince0656,

> Just curious: why revert at all? There's no harm in writing several times in storage `isPriorityStaker[staker]` = true with a wrong input. However, "checking `isPriorityStaker[staker]` and revert if true" will penalize every caller as all these storage reading operations are expensive.

> I don't believe the happy path should cost more gas just to prevent an unhappy one.

> But that's really just a suggestion on the remediation.

> Edit:
> I'll also add here that a DOS (it can be worked around here but this is still a

> degraded functionality, with a damaged availability) is Medium Severity usually on code4rena's documentation, not low, due to "the function of the protocol or its availability could be impacted":

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> I also ask here if you could please reconsider this as just Medium 👍. Of course, I'll accept any final decision you make.

teryanarmen (Certora) commented:

> I think there are two separate issues here. I believe the failure to check for duplicate entries is Low/Informational severity as having duplicates in the `_priorityStakers` array has no effect on the protocol. The check being unnecessary and causing a DOS to me is medium severity.

[M-02]

Submitted by Koolex

Summary

Deregistering inactive knot via calling deRegisterKnots directly by the Syndicate owner always reverts.

Description

Actual Behaviour & Impact

The liquid staking manager (Syndicate owner) calls `deRegisterKnots` function to deregister knots. However, if any of those knots is inactive, the transaction will revert. This means in order to deregister an inactive knot, one should call `updateCollateralizedSlotOwnersAccruedETH` function since it calls internal `_deRegisterKnots`.

Expected Behaviour

Deregistering knots by Syndicate owner should succeed especially if the knot is inactive.

Cause

There is a circular dependency between deRegisterKnots and `_updateCollateralizedSlotOwnersLiabilitySnapshot` functions which leads to this behaviour.

`deRegisterKnots` always calls internally `_updateCollateralizedSlotOwnersLiabilitySnapshot`, and `_updateCollateralizedSlotOwnersLiabilitySnapshot` calls `_deRegisterKnot` in case the knot is inactive. So when the knot is deregistered the first time, the second time it reverts.

https://github.com/koolexcrypto/2023-01-blockswap-fv-private/blob/certora/contracts/syndicate/Syndicate.sol#L571

https://github.com/koolexcrypto/2023-01-blockswap-fv-private/blob/certora/contracts/syndicate/Syndicate.sol#L641

Property Violated

`isNoLongerPartOfSyndicate` is not set to true since the function reverts.

Detected by

Rule `deregisterInactiveKnotShouldSucceed()`

vince0656 (Blockswap) commented:

> Assessment: Medium/High

> We have made sure the logic will not revert when deregistering an inactive knot.

teryanarmen (Certora) commented:

> Since inactive knots can still be deregistered by calling updateCollateralizedSlotOwnersAccruedETH, I believe this is medium severity.

[M-03]

Submitted by jessicapointing, also found by Koolex

Description of the problem

A comment in Syndicate.sol states that “Basically, under a rage quit or voluntary withdrawal from the beacon chain, the knot kick is auto-propagated to syndicate”. However, when a KNOT is inactive, the KNOT can still remain part of the syndicate and isn’t automatically removed from the syndicate.

Expected Behaviour

After calling any function, an `inActive` KNOT should become no longer part of the syndicate.

Actual behaviour of the system

An inactive KNOT can still be part of the syndicate.

In this rule, the `isActive` state of the KNOT is set to false and a function is called and the `isNoLongerPartOfSyndicate` state is checked.

Property Violated: `inactiveKnotShouldNoLongerBePartOfSyndicate`

vince0656 (Blockswap) commented:

> Our assessment: Medium severity

> We will expose a function for anyone to poke the syndicate when a knot becomes inactive which will then kick them from the syndicate.

teryanarmen (Certora) commented:

> This seems out of scope as a knot is made inactive outside of the syndicate and we should assume that the external contract will deregister the appropriate knot when they are made inactive.

teryanarmen (Certora) commented:

> Actually since the protocol doesn't poke Syndicate directly and Syndicate is responsible for deregistering knots once they are inActive, failing to do so can lead to value being leaked from active and registered knots on a syndicate to inactive and registered knots. Since this leak is stopped when collateralizedSLOT owners claim fees and can be triggered freely by anyone, the bug is medium severity.

[M-04]

Submitted by eighty

Short description

Some rewards for free floating shares users could become unclaimable if the user is not careful when unstaking from a knot.

Elaborative explanation of the bug and an attack case example

When staking on a given knot, the protocol prevents staking amounts fewer than 1 gwei. Additionally, when unstaking, there's a check against unstaking amounts greater than the amounts allowed (at 263), but there's no check on the remaining amount. Thus, this leaves open scenarios where the user may leave less than 1 gwei on a knot. In such cases, the user cannot claim additional earnings due to the following calls: 679 - 693 - 370.

Additional reasoning

The user could "restake" his position on a knot, but there's a (short) limit of 12 eth that could be filled by anyone.
A Knot could become inactive, making future rewards forever inaccessible.
The user could recover the smalls staked amounts, but he'll not be compensated by accrued earnings.
Property violated.

Rule `issue2_unstakingLeavesSmallAmountsBehind` in `M002.spec` , regarding the possibility of small stakes.
Rule `issue2_ifTheUserHasClaimableAmountsHeShouldBeAbleToClaimIt` in `M002.spec` regarding the rewards unclaimable.

Mitigation

Prevent unstaking shares when the amount remaining is less than 1 gwei.

vince0656 (Blockswap) commented:

> Assessment: Low/Medium

> We will remove the check on the minimum amount that must be unstaked to avoid this. Thanks.

teryanarmen (Certora) commented:

> I believe this is medium severity as the protocol leaks funds but major funds are not at risk.

[M-05]

Submitted by jessicapointing

Description of the problem

An inactive KNOT can successfully stake.

Expected behaviour of the system: The stake function should revert if an inactive KNOT is trying to stake.

Actual behaviour of the system: The stake function succeeds and the inactive KNOT stakes.

Property Violated: `cannotStakeIfKnotIsInActive`

In this rule, the `isActive` state of the `blsPubKey` is set to `false`. The stake function is then called with this `blsPubKey` and the expected behaviour is that the function would revert because the KNOT is inactive. However, the stake function succeeds and an inactive KNOT can therefore successfully stake. The stake function checks if KNOT is not registered (`!isKNOTRegistered`) and if it is no longer part of the syndicate (`!isNoLongerPartOfSyndicate`) and reverts accordingly but does not check whether the KNOT is inactive.

Lines of code: https://github.com/Certora/2023-01-blockswap-fv/blob/certora/contracts/syndicate/Syndicate.sol#L216

Potential Fix

```
    (,,,,,bool isActive) = getStakeHouseUniverse().stakeHouseKnotInf
    if (!isKnotRegistered[_blsPubKey] || isNoLongerPartOfSyndicate[_
```

jessicapointing (warden) commented:

The principle found here can be applied to other functions in the code which reveal potentially more bugs. In other words, just as I have pointed out here that an `inActive` KNOT can successfully call stake and succeed, an `inActive` KNOT (or `!isKnotRegistered` or `isNoLongerPartOfSyndicate` KNOT) can successfully call other functions it shouldn't be able to and succeed. To fix these bugs, check the `isActive` / `isKnotRegistered` / `isNoLongerPartOfSyndicate` state of a KNOT before executing the function. Maybe the lack of checks are intentional because inactive/deregistered knots can still call some functions but I'm including them here for completeness and in case they shouldn't be able to execute such functions. Here are some of the other functions and lines of code:

[`unstake`]():
`isActive` and `isKnotRegistered` is not checked.

[`claimAsCollateralizedSLOTOwner`]():
`isActive` and `isNoLongerPartOfSyndicate` is not checked.

[`_updateCollateralizedSlotOwnersLiabilitySnapshot`]():
`isActive` and `isKnotRegistered` is not checked.

[`_claimAsStaker`]():
`isActive` and `isNoLongerPartOfSyndicate` is not checked.

vince0656 (Blockswap) commented:

> Assessment: Low severity because if you stake without an active knot, you don't earn rewards but you can always get your stake back.

> We will check for active status when staking - thanks.

teryanarmen (Certora) commented:

> Inactive knots can earn rewards since all rewards are compiled together, but since deregistering a knot is done when the collateralized staker claims rewards and can be triggered at any time by anyone I think this is medium severity.

[M-06]

Submitted by neumo

Summary

Whenever a knot is deregistered, the last value of `accumulatedETHPerFreeFloatingShare` is stored in the mapping `lastAccumulatedETHPerFreeFloatingShare`. The accumulated value must be updated always before deregistering the knot, otherwise stakers of the knot could receive less ETH than they should when claiming/unstaking. A rule allowed me to detect two functions that violate this property.

Explanation

Function `_deRegisterKnot` is in charge of marking a knot as `isNoLongerPartOfSyndicate`:

```
    /// @dev Business logic for de-registering a specific knots assu
    function _deRegisterKnot(bytes memory _blsPublicKey) internal {
            if (isKnotRegistered[_blsPublicKey] == false) revert Knc
            if (isNoLongerPartOfSyndicate[_blsPublicKey] == true) re

            // We flag that the knot is no longer part of the syndic
            isNoLongerPartOfSyndicate[_blsPublicKey] = true;
```

```
        // For the free floating and collateralized SLOT of the
        lastAccumulatedETHPerFreeFloatingShare[_blsPublicKey] =

        // We need to reduce `totalFreeFloatingShares` in order
        totalFreeFloatingShares -= sETHTotalStakeForKnot[_blsPuk

        // Total number of registered knots with the syndicate r
        numberOfRegisteredKnots -= 1;

        emit KnotDeRegistered(_blsPublicKey);
    }
```

We can see it snapshots the current value of `accumulatedETHPerFreeFloatingShare` in the mapping `lastAccumulatedETHPerFreeFloatingShare`. This is because all claiming/unstaking on an unregistered knot should take into account the accrued ETH up to the time of deregistering. So it means, that `accumulatedETHPerFreeFloatingShare` must be up to date every time a knot is deregistered.

I wrote the following rule to test if this property holds after every call of the contract:

```
    rule lastAccumulatedETHPerFreeFloatingShareMustAccountForAccruec
        f -> notHarnessCall(f)
    }{

        env e;

        bytes32 blsPubKey;

        require isKnotRegistered(blsPubKey);
        require !isNoLongerPartOfSyndicate(blsPubKey);
        require lastAccumulatedETHPerFreeFloatingShare(blsPubKey) ==

        calldataarg args;
        f(e, args);

        require isNoLongerPartOfSyndicate(blsPubKey);

        updateAccruedETHPerShares(e);

        assert lastAccumulatedETHPerFreeFloatingShare(blsPubKey) ==
    }
```

Basically what it does is:

- Ensure that the knot with `blsPubKey` is registered
- Ensure that it is part of Syndicate
- Ensure that `lastAccumulatedETHPerFreeFloatingShare` for the knot is zero.
- Execute a call to any function of the contract
- Ensure that after the call, the knot has been deregistered (`isNoLongerPartOfSyndicate = true`)
- Call update accrued ETH per shares
- Assert that `lastAccumulatedETHPerFreeFloatingShare` of the knot is equal to `accumulatedETHPerFreeFloatingShare`

If the rule fails, we can affirm that `accumulatedETHPerFreeFloatingShare` was not up to date, because the call to `updateAccruedETHPerShares` should not change its value.

But I found calls to these two functions make the rule fail:

- `updateCollateralizedSlotOwnersAccruedETH`
- `batchUpdateCollateralizedSlotOwnersAccruedETH`

That is because these two functions don't call `updateAccruedETHPerShares` and they call `_updateCollateralizedSlotOwnersLiabilitySnapshot`, which also does not call the update accrued ETH function. But this last function can deregister the knot (if it is not active in Stakehouse, see https://github.com/Certora/2023-01-blockswap-fv/blob/certora/contracts/syndicate/Syndicate.sol#L570-L572).

The impact of this issue is the loss of part of the rewards for the affected stakers, because their share of rewards will be calculated with an old value of the accrued rewards, which is for sure less than it should be (as accumulatedETHPerFreeFloatingShare cannot decrease, as I checked in another rule).

Impact

High impact, because it implies loss of funds for stakers of the protocol.

Property violated

After a knot state goes from `isNoLongerPartOfSyndicate[blsPubKey] == false` to `isNoLongerPartOfSyndicate[blsPubKey] == true` the value stored at `lastAccumulatedETHPerFreeFloatingShare[blsPubKey]` should be always equal to the up to date value of `accumulatedETHPerFreeFloatingShare`.

Recommendation

Call `updateAccruedETHPerShares` inside functions `updateCollateralizedSlotOwnersAccruedETH` and `batchUpdateCollateralizedSlotOwnersAccruedETH`:

```
    function updateCollateralizedSlotOwnersAccruedETH(blsKey _blsPuk
            updateAccruedETHPerShares();
            _updateCollateralizedSlotOwnersLiabilitySnapshot(_blsPuk
    }

    function batchUpdateCollateralizedSlotOwnersAccruedETH(blsKey[]
            uint256 numOfKeys = _blsPubKeys.length;
            if (numOfKeys == 0) revert EmptyArray();
            updateAccruedETHPerShares();
            for (uint256 i; i < numOfKeys; ++i) {
                    _updateCollateralizedSlotOwnersLiabilitySnapshot
            }
    }
```

vince0656 (Blockswap) commented:

> Assessment: Medium

> Thank you - we have fixed this issue in the Syndicate contract.

teryanarmen (Certora) commented:

> In the case that a knot successfully validates a block and earns rewards, becomes inactive and is deregistered through `updateCollateralizedSlotOwnersAccruedETH` or `batchupdateCollateralizedSlotOwnersAccruedETH`, all without any other interaction with Syndicate which calls `updateAccruedETHPerShares` method, the the stakers of the knot would not be able to claim these rewards and further those funds will be locked in the contract. Since this vulnerability leads to non-negligible loss of funds I believe it is high severity.

> One comment, @neumo why did you choose to add

> require `lastAccumulatedETHPerFreeFloatingShare(blsPubKey) == 0;`

> instead of just storing the output of `lastAccumulatedETHPerFreeFloatingShare(blsPubKey)` in a variable before the call to `updateAccruedETHPerShares` and asserting that the variable was equal to `accumulatedETHPerFreeFloatingShare()`? Seems to unnecessarily limit the scope.

neumo (warden) commented:

> @teryanarmen - My reasoning was to assure that before calling the function the value of `lastAccumulatedETHPerFreeFloatingShare` was 0 because that is the value registered knots have. In fact its value is only set in one place (function `_deRegisterKnot`), so I guessed it was safe (and less computationally intensive for the rule) to assume it to be zero before the call. Your approach is also valid too, of course. And even removing the require should work too because, ultimately, the value before the call does not matter, what is important is that after the call to `updateAccruedETHPerShares` the assert still holds.

> I hope I answered your question!

teryanarmen (Certora) commented:

> I actually think this is a medium severity since assets can't be lost or stolen directly and have some external dependencies such as validating a block at the right time and no one calling `updateAccruedETHPerShares`.

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 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