code4rena

Sherlock contest
Findings & Analysis Report

2022-05-17

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 audit 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 audit contest outlined in this document, C4 conducted an analysis of the Sherlock smart contract system written in Solidity. The audit contest took place between January 20—January 26 2022.

WARDENS

40 Wardens contributed reports to the Sherlock contest:

1. GreyArt (hickuphh3 and itsmeSTYJ)
2. OriDabush
3. egjlmn1
4. kirk-baird
5. pauliax
6. hyh
7. static
8. sirhashalot
9. ye0lde
10. danb
11. hack3r-0m
12. cccz
13. harleythedog
14. kenzo

15. Dravee
16. 0x0x0x
17. Jujic
18. 0x1f8b
19. RamiRond
20. gzeon
21. Tomio
22. bobi
23. Fitraldys
24. haku
25. robee
26. Funen
27. Afanasyevich
28. p4st13r4 (0x69e8 and 0xb4bb4)
29. wuwe1
30. defsec
31. Ruhum
32. pedroais
33. ych18
34. byterocket (pseudorandom and pmerkleplant)
35. saian
36. ACai
37. GeekyLumberjack

This contest was judged by Jack the Pug.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 14 unique vulnerabilities and 71 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 9 received a risk rating in the category of LOW severity.

C4 analysis also identified 33 non-critical recommendations and 24 gas optimizations.

Scope

The code under review can be found within the C4 Sherlock contest repository, and is composed of 8 smart contracts written in the Solidity programming language and includes 612 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 (1)

[H-01] FIRST USER CAN STEAL EVERYONE ELSE'S TOKENS

Submitted by egjlmn1, also found by OriDabush

A user who joins the systems first (stakes first) can steal everybody's tokens by sending tokens to the system externally. This attack is possible because you enable staking a small amount of tokens.

Proof of Concept

See the following attack:

1. the first user (user A) who enters the system stake 1 token
2. another user (user B) is about to stake X tokens
3. user A frontrun and transfer X tokens to the system via `ERC20.transfer`
4. user B stakes X tokens, and the shares he receives is:

`shares = (_amount * totalStakeShares_) / (totalTokenBalanceStakers() - _amount);`
`shares = (X * 1) / (X + 1 + X - X) = X/(X+1) = 0` meaning all the tokens he staked got him no shares, and those tokens are now a part of the single share that user A holds 5. user A can now redeem his shares and get the 1 token he staked, the X tokens user B staked, and the X tokens he `ERC20.transfer` to the system because all the money in the system is in a single share that user A holds.

In general, since there is only a single share, for any user who is going to stake X tokens, if the system has X+1 tokens in its balance, the user won't get any shares and all the money will go to the attacker.

Recommended Mitigation Steps

Force users to stake at least some amount in the system (Uniswap forces users to pay at least `1e18`) That way the amount the attacker will need to ERC20.transfer to the system will be at least `X*1e18` instead of `X` which is unrealistic

Evert0x (Sherlock) confirmed and commented:

> Thanks. I agree it's an issue that could theoretically affect all deposits.

Evert0x (Sherlock) resolved

Medium Risk Findings (4)

[M-01] SHERLOCKCLAIMMANAGER: INCORRECT AMOUNTS NEEDED AND PAID FOR ESCALATED CLAIMS

Submitted by GreyArt, also found by static

When escalating claims, the documentation states that the protocol agent is required to pay and stake a certain amount for the process. If the covered protocol is proven correct, then the amount specified by the claim will be paid out. They will also receive the stake amount back in full. If the covered protocol's escalation is not successful, then the amount specified by the claim is not paid out and the stake amount is not returned.

The protocol agent is reasonably expected to pay the following:

- The stake (`BOND`) and
- UMA's final fee

In reality, the protocol agent will end up paying more, as we shall see in the proof of concept.

Proof of Concept

Let us assume the following:

- `BOND = 9600` as defined in `SherlockClaimManager`
- `umaFee = 400` (at the time of writing, this value has been updated to 1500 USDC: see `[Store.computeFinalFee(usdc)](https://etherscan.io/address/0x54f44eA3D2e7aA0ac089c4d8F7C93C27844057BF#readContract)`).

On invoking `escalate()`, the following amounts are required:

1. `BOND + umaFee = 9600 + 400` will be transferred to UMA when invoking `requestAndProposePriceFor()`

```
// Taken from https://github.com/UMAprotocol/protocol/blob/master/packages/core/contract
// Only relevant lines are referenced
uint256 finalFee = _getStore().computeFinalFee(address(currency)).rawValue;
request.finalFee = finalFee;
totalBond = request.bond.add(request.finalFee);
if (totalBond > 0) currency.safeTransferFrom(msg.sender, address(this), totalBond);
```

1. Another `BOND + umaFee = 9600 + 400` will be transfered when invoking `disputePriceFor()`

```
390
391  totalBond = request.bond.add(request.finalFee);
392  if (totalBond > 0) request.currency.safeTransferFrom(msg.sender, address(this), tota
```

However, what's important to note is that UMA will "burn" half of the BOND collected + final fee. This will go against the claim that the protocol agent will be able to reclaim his stake in full.

```
StoreInterface store = _getStore();

// Avoids stack too deep compilation error.
{
    // Along with the final fee, "burn" part of the loser's bond to ensure that a larger
    // proportionally more expensive to delay the resolution even if the proposer and di
    // party.
    uint256 burnedBond = _computeBurnedBond(disputedRequest);

    // The total fee is the burned bond and the final fee added together.
    uint256 totalFee = request.finalFee.add(burnedBond);

    if (totalFee > 0) {
        request.currency.safeIncreaseAllowance(address(store), totalFee);
        _getStore().payOracleFeesErc20(address(request.currency), FixedPoint.Unsigned(to
    }
}

function _computeBurnedBond(Request memory request) private pure returns (uint256) {
  // burnedBond = floor(bond / 2)
  return request.bond.div(2);
}
```

We finally note that on settlement, the eventual payout is

```
// Winner gets:
// - Their bond back.
// - The unburned portion of the loser's bond: proposal bond (not including final fee) -
// - Their final fee back.
// - The request reward (if not already refunded -- if refunded, it will be set to 0).
payout = request.bond.add(request.bond.sub(_computeBurnedBond(settledRequest))).add(requ
    request.reward
);
request.currency.safeTransfer(disputeSuccess ? request.disputer : request.proposer, payo
```

Hence, in reality, the protocol agent will only receive `9600 * 2 - 4800 + 400 = 14800` should the dispute be successful. We note that the burnt amount of `4800 / 2 + 400 = 5200` has been taken by UMA.

One can further verify this behaviour by looking at a past resolution of another protocol:

https://dashboard.tenderly.co/tx/main/0x0f03f73a2093e385146791e8f2739dbc04b39145476d6940776680243460100f trace=0.6.1

The above example has a bond is `0.0075 ETH`, with UMA's final fee being 0.2 ETH. We see that UMA takes `0.2 + 0.5 * 0.0075 = 0.02375 ETH`.

Thus, we see that the protocol agent will be charged disproportionally to what is expected.

Recommended Mitigation Steps

We suggest changing the parameters of `requestAndProposePriceFor()` to

```
UMA.requestAndProposePriceFor(
  UMA_IDENTIFIER, // Sherlock ID so UMA knows the request came from Sherlock
  claim.timestamp, // Timestamp to identify the request
  claim.ancillaryData, // Ancillary data such as the coverage agreement
  TOKEN, // USDC
  BOND, // While sherlock handles rewards on its own, we use the BOND as the reward
  // because using it as UMA's bond would result in 0.5 * BOND charged by UMA excluding
  1, // Ideally 0, but 0 = final fee used. Hence, we set it to the next lowest
  // possible value
  LIVENESS, // Proposal liveness
  address(sherlockCore), // Sherlock core address
  0 // price
);
```

where `BOND` becomes the reward and the actual bond for UMA is `1`. Ideally, it should be set to 0, but if set as such, UMA interprets it to use the final fee as the bond amount instead.

```
[request.bond = bond != 0 ? bond : finalFee;]
(https://github.com/UMAprotocol/protocol/blob/master/packages/core/contracts/oracle/implementation/SkinnyOptir
```

This way, the actual amount required from the protocol agent is the `BOND + 2 * (USDC wei + umaFee)` for the process. He will additionally be returned his `BOND + umaFee` if his dispute is successful.

Evert0x (Sherlock) disagreed with High severity and commented:

> Non critical as documentation is incorrect about this.

Jack the Pug (judge) decreased severity to Medium and commented:

> Downgrading to `Med` as it's mostly because the documentation is incorrect.

rcstanciu (Sherlock) commented:

> @jack-the-pug The docs have been updated to explain the correct burned amount.

Jack the Pug (judge) commented:

Thank you @rcstanciu

While I agreed that the issue only impacts a small sum of users and the impact is not significant. And the root cause of this issue may not be a wrong implementation but actual wrong documentation.

However, I still tend to make this a `Med` rather than a `Low` for the following reasons:

1. A wrong documentation is arguably indistinguishable from a wrong implementation that actually violates the intention of the design, especial from an outsider's pov;
2. This write-up indicates a dedicated and in-depth effort of the warden by digging into the documentation and trying to understand the intention and cross-compare with the actual implementation to find any differences. As a judge, part of my job is to make sure that the wardens' findings are being rewarded in a just and fair manner.

Therefore, I'm keeping this as a `Med` and I encourage the future wardens to continue finding the inconsistency between the documentation and the implementation.

Keep up the good work! GreyArt is a great name btw.

[M-02] `tokenBalanceOfAddress` of `nftOwner` becomes permanently incorrect after `arbRestake`

Submitted by hyh, also found by GreyArt and hack3r-0m

Successful `arbRestake` performs `_redeemShares` for `arbRewardShares` amount to extract the arbitrager reward. This effectively reduces shares accounted for an NFT, but leaves untouched the `addressShares` of an `nftOwner`.

As a result the `tokenBalanceOfAddress` function will report an old balance that existed before arbitrager reward was slashed away. This will persist if the owner will transfer the NFT to someone else as its new reduced shares value will be subtracted from `addressShares` in `_beforeTokenTransfer`, leaving the arbitrage removed shares permanently in `addressShares` of the NFT owner, essentially making all further reporting of his balance incorrectly inflated by the cumulative arbitrage reward shares from all arbRestakes happened to the owner's NFTs.

Proof of Concept

`arbRestake` redeems `arbRewardShares`, which are a part of total shares of an NFT:

Sherlock.sol#L673

This will effectively reduce the `stakeShares`:

Sherlock.sol#L491

But there is no mechanics in place to reduce `addressShares` of the owner apart from mint/burn/transfer, so `addressShares` will still correspond to NFT shares before arbitrage. This discrepancy will be accumulated further with arbitrage restakes.

Recommended Mitigation Steps

Add a flag to `_redeemShares` indicating that it was called for a partial shares decrease, say `isPartialRedeem`, and do `addressShares[nftOwner] -= _stakeShares` when `isPartialRedeem == true`.

Another option is to do bigger refactoring, making stakeShares and addressShares always change simultaneously.

Evert0x (Sherlock) confirmed and commented:

> This is a legit issue and needs to be addressed. I think we choose to delete this functionality all together.
>
> The function has some potential future benefit but it might be too little benefit to make these relatively complex changes that make the code harder to understand.

Evert0x (Sherlock) resolved

[M-03] `updateYieldStrategy` will freeze some funds with the old strategy if `yieldStrategy` fails to withdraw all the funds because of liquidity issues

Submitted by hyh, also found by harleythedog, GreyArt, and pauliax

Part of the funds held with the strategy can be frozen if the current strategy has tight liquidity when `updateYieldStrategy` is run as this function makes an attempt to withdraw all the funds and then unconditionally removes the strategy.

The Sherlock to YieldStrategy link will be broken as a result: Sherlock points to the new Strategy, while old Strategy still allows only this Sherlock contract to withdraw.

This way back and forth switches will be required in the future to return the funds: withdraw all from new strategy and switch to old, withdraw all from old and point to new one again, reinvest there.

Proof of Concept

In peer-to-peer lending protocols it is not always possible for the token supplier to withdraw all what's due. This happens on high utilization of the market (when it has a kind of liquidity crunch).

This way yieldStrategy.withdrawAll is not guaranteed to obtain all the funds held with the strategy:

Sherlock.sol#L286

The worst case scenario here seems to be the remainder funds to be left frozen within the strategy.

For example, AaveV2Strategy `withdraw` and `withdrawAll` have `onlySherlockCore` modifier:

AaveV2Strategy.sol#L78-100

While Sherlock core is immutable for the Strategy by default:

Manager.sol#L26-41

Recommended Mitigation Steps

Consider implementing a new method that fails whenever a strategy cannot withdraw all what's due now, and rename current implementation to, for example, `forceUpdateYieldStrategy`, to have a degree of flexibility around liquidity issues.

Also, to avoid back and forth switching, a strategy argument can be introduced to `yieldStrategyWithdrawAll` to allow withdrawals from any (not only current) yieldStrategy:

Sherlock.sol#L322

Now:

```
function yieldStrategyWithdrawAll() external override onlyOwner {
```

To be (if `_yieldStrategy` is zero then utilize current):

```
function yieldStrategyWithdrawAll(IStrategyManager _yieldStrategy) external override onl
```

Evert0x (Sherlock) disagreed with Medium severity and commented:

> I think this should be low risk but it's an interesting feature to add

Jack the Pug (judge) commented:

> I think this worths a `Med`, the scenario is not impossible to happen, and the handling in the current implementation is quite rough.

[M-04] REENTERANCY IN `_SENDSHERREWARDSTOOWNER()`

Submitted by kirk-baird

This is a reentrancy vulnerability that would allow the attacker to drain the entire SHER balance of the contract.

Note: this attack requires gaining control of execution `sher.transfer()` which will depend on the implementation of the SHER token. Control may be gained by the attacker if the contract implements ERC777 or otherwise makes external calls during `transfer()`.

Proof of Concept

See _sendSherRewards

```
function _sendSherRewardsToOwner(uint256 _id, address _nftOwner) internal {
  uint256 sherReward = sherRewards_[_id];
  if (sherReward == 0) return;

  // Transfers the SHER tokens associated with this NFT ID to the address of the NFT c
  sher.safeTransfer(_nftOwner, sherReward);
  // Deletes the SHER reward mapping for this NFT ID
  delete sherRewards_[_id];
```

```
}
```

Here `sherRewards` are deleted after the potential external call is made in `sher.safeTransfer()`. As a result if an attacker reenters this function `sherRewards_` they will still maintain the original balance of rewards and again transfer the SHER tokens.

As `_sendSherRewardsToOwner()` is `internal` the attack can be initiated through the `external` function `ownerRestake()` see here.

Steps to produce the attack:

1. Deploy attack contract to handle reenterancy
2. Call `initialStake()` from the attack contract with the smallest `period`
3. Wait for `period` amount of time to pass
4. Have the attack contract call `ownerRestake()`. The attack contract will gain control of the (See note above about control flow). This will recursively call `ownerRestake()` until the balance of `Sherlock` is 0 or less than the user's reward amount. Then allow reentrancy loop to unwind and complete.

Recommended Mitigation Steps

Reentrancy can be mitigated by one of two solutions.

The first option is to add a reentrancy guard like `nonReentrant` the is used in `SherlockClaimManager.sol`.

The second option is to use the checks-effects-interactions pattern. This would involve doing all validation checks and state changes before making any potential external calls. For example the above function could be modified as follows.

```
function _sendSherRewardsToOwner(uint256 _id, address _nftOwner) internal {
  uint256 sherReward = sherRewards_[_id];
  if (sherReward == 0) return;

  // Deletes the SHER reward mapping for this NFT ID
  delete sherRewards_[_id];

  // Transfers the SHER tokens associated with this NFT ID to the address of the NFT c
  sher.safeTransfer(_nftOwner, sherReward);
}
```

Additionally the following functions are not exploitable however should be updated to use the check-effects-interations pattern.

- `Sherlock._redeemShares()` should do `_transferTokensOut()` last.
- `Sherlock.initialStake()` should do `token.safeTransferFrom(msg.sender, address(this), _amount);` last
- `SherClaim.add()` should do `sher.safeTransferFrom(msg.sender, address(this), _amount);` after updating `userClaims`

- `SherlockProtocolManager.depositToActiveBalance()` should do `token.safeTransferFrom(msg.sender, address(this), _amount);` after updating `activeBalances`

Evert0x (Sherlock) confirmed, but disagreed with High severity and commented:

> Good find. I think it's med-risk as it depends on the implementation of SHER token (does it allow callbacks).

Jack the Pug (judge) decreased severity to Medium and commented:

> Downgrade to `Med` as the SHER token is a known token that currently comes with no such hooks like ERC777.

Evert0x (Sherlock) resolved

Low Risk Findings (9)

- [L-01] safeApprove will fail if the current approval is not 0 *Submitted by pauliax, also found by cccz and sirhashalot*
- [L-02] Slippage parameter for SherBuy *Submitted by pauliax*
- [L-03] Many `protocolUpdate()` calls erase historic previousCoverage *Submitted by sirhashalot*
- [L-04] SherBuy: SHER and USDC token addresses should be derived from _sherlockPosition *Submitted by GreyArt, also found by cccz*
- [L-05] Sherlock: Revert for non-existent ID in viewRewardForArbRestake *Submitted by GreyArt*
- [L-06] Wrong user input check of incoming USDC when escalating claim *Submitted by kenzo, also found by GreyArt*
- [L-07] missing whenNotPaused *Submitted by danb*
- [L-08] Pausable paused() is not enforced to be present *Submitted by pauliax*
- [L-09] Missing parameter check or confusing comment in function protocolRemove (SherlockProtocolManager.sol) *Submitted by ye0lde*

Non-Critical Findings (33)

- [N-01] No Transfer Ownership Pattern *Submitted by cccz, also found by Afanasyevich and Dravee*
- [N-02] USDC is upgradeable: received amount should be calculated *Submitted by Dravee, also found by harleythedog*
- [N-03] There is a deviation in the parameter value setting. *Submitted by ACai*
- [N-04] Use of the reserved keyword `error` as a variable name *Submitted by Dravee*
- [N-05] `SherlockProtocolManager.sol`: `setMinActiveBalance()` and `forceRemoveByActiveBalance()` should be put behind a timelock *Submitted by Dravee*
- [N-06] _isEscalateState Comment Improvement *Submitted by GeekyLumberjack*

- [N-07] Withrawals will fail if the market has high utilization *Submitted by pedroais*
- [N-08] Claim SHER on behalf of others *Submitted by pauliax*
- [N-09] Re-entrancy protection *Submitted by pauliax*
- [N-10] safeApprove of openZeppelin is deprecated *Submitted by robee, also found by bobi, cccz, and byterocket*
- [N-11] Spelling Errors *Submitted by GreyArt, also found by 0x0x0x and p4st13r4*
- [N-12] Bond price not controlled by Sherlock *Submitted by sirhashalot, also found by kenzo*
- [N-13] SherlockClaimManager: reentrancy comment for priceProposed() and priceDisputed() can be phrased better *Submitted by GreyArt*
- [N-14] Updating Manager contract could destruct Sherlock core functionalities *Submitted by ych18*
- [N-15] Unnecessary typcasting *Submitted by saian*
- [N-16] Never-used ETH transfers in _sweep *Submitted by p4st13r4*
- [N-17] Use structs instead of three mappings in Sherlock.sol *Submitted by p4st13r4*
- [N-18] Inconsistent Acronym of UmaHaltOperator *Submitted by GreyArt*
- [N-19] Grammar *Submitted by GreyArt*
- [N-20] SherlockClaimManager: Clarify why sherlockCore is used as proposer in UMA.requestAndProposePriceFor() *Submitted by GreyArt*
- [N-21] SherlockClaimManager: startClaim() has outdated comment *Submitted by GreyArt*
- [N-22] SherlockClaimManager: Confusing comment on BOND *Submitted by GreyArt*
- [N-23] ISherlockClaimManager: Outdated example on claims process *Submitted by GreyArt*
- [N-24] Name collision in `SherlockProtocolManager` *Submitted by gzeon*
- [N-25] yieldStrategyDeposit doesn't check that there is enough USDC to deposit *Submitted by hyh*
- [N-26] Incorrect comment about callback call *Submitted by sirhashalot*
- [N-27] ISherlockGov.removeSherDistributionManager description is incorrect *Submitted by hyh*
- [N-28] Implementation is not align with documentation *Submitted by wuwe1*
- [N-29] Implementation is not align with documentation #2 *Submitted by wuwe1*
- [N-30] Pause/unpause functions descriptions aren't fully correct *Submitted by hyh*
- [N-31] addressShares introduction made further shares accounting error prone *Submitted by hyh*
- [N-32] Incorrect comparison for prevCoverage in startClaim() *Submitted by GeekyLumberjack*
- [N-33] Hardhat references in Manager.sol code *Submitted by p4st13r4*

Gas Optimizations (24)

- [G-01] Adding unchecked directive can save gas *Submitted by defsec, also found by bobi, Dravee, ye0lde, Afanasyevich, OriDabush, Jujic, and pauliax*
- [G-02] Gas: `++i` costs less gas compared to `i++` *Submitted by Dravee, also found by robee, p4st13r4, Afanasyevich, Funen, OriDabush, defsec, and pedroais*

- [G-03] Cache array length in for loops can save gas *Submitted by defsec, also found by robee, 0x0x0x, bobi, Dravee, OriDabush, byterocket, Jujic, GreyArt, gzeon, saian, and Fitraldys*
- [G-04] Avoid unneeded SLOADs by caching values in memory *Submitted by RamiRond, also found by kirk-baird, p4st13r4, Ruhum, and Dravee*
- [G-05] `SherDistributionManager.sol#slopeRewardsAvailable` can be calculated later to save gas *Submitted by 0x0x0x*
- [G-06] Gas in `SherlockClaimManager.sol:priceDisputed():` a value used only once shouldn't get cached *Submitted by Dravee*
- [G-07] Saving gas by used length *Submitted by Funen, also found by 0x1f8b and Tomio*
- [G-08] Cheaper to use calldata than memory *Submitted by Tomio, also found by Jujic and robee*
- [G-09] Gas: Using the logical NOT operator `!` is cheaper than a comparison to the constant boolean value `false` *Submitted by Dravee, also found by 0x1f8b, 0x0x0x, ych18, ye0lde, haku, GreyArt, and Fitraldys*
- [G-10] 10 ** 18 can be changed to 1e18 and save some gas *Submitted by Jujic*
- [G-11] Manager: Check non-zero ETH balance before sending *Submitted by GreyArt, also found by Tomio*
- [G-12] SherDistributionManager: Cheaper to assign than add _tvl *Submitted by GreyArt*
- [G-13] `Sherlock.sol#_beforeTokenTransfer()` has not needed if statements *Submitted by 0x0x0x*
- [G-14] gas saving III *Submitted by 0x1f8b*
- [G-15] If condition can be removed from SherlockClaimManager.cleanUp function *Submitted by Afanasyevich, also found by ye0lde, sirhashalot, and wuwe1*
- [G-16] Gas: "constants" expressions are expressions, not constants. Use "immutable" instead. *Submitted by Dravee, also found by Jujic*
- [G-17] Gas: Consider making some constants as non-public to save gas *Submitted by Dravee, also found by haku*
- [G-18] cheaper to calculate stakeShares first then do the transfer *Submitted by Fitraldys, also found by OriDabush*
- [G-19] saving gas on reverted transactions *Submitted by OriDabush*
- [G-20] Use return value of assignments to save gas *Submitted by RamiRond*
- [G-21] Function call can be done after required check. *Submitted by bobi, also found by pauliax*
- [G-22] Gas Optimization: Struct layout *Submitted by gzeon*
- [G-23] debt = balance *Submitted by pauliax, also found by Dravee*
- [G-24] Gas: Non-strict inequalities are cheaper than strict ones *Submitted by Dravee*

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.

TWITTER // DISCORD // GITHUB

0XC2BC2F890067C511215F9463A064221577A53E10 //