



Backed Protocol contest Findings & Analysis Report

2022-04-21

Table of contents

- [Overview](#)
 - [About C4](#)
 - [Wardens](#)
- [Summary](#)
- [Scope](#)
- [Severity Criteria](#)
- [High Risk Findings \(3\)](#)
 - [\[H-01\] Can force borrower to pay huge interest](#)
 - [\[H-02\] currentLoanOwner can manipulate loanInfo when any lenders try to buyout](#)
 - [\[H-03\] Borrower can be their own lender and steal funds from buyout due to reentrancy](#)
- [Medium Risk Findings \(7\)](#)
 - [\[M-01\] When an attacker lends to a loan, the attacker can trigger DoS that any lenders can not buyout it](#)
 - [\[M-02\] Protocol doesn't handle fee on transfer tokens](#)

- [M-03] `sendCollateralTo` is unchecked in `closeLoan()`, which can cause user's collateral NFT to be frozen
- [M-04] `requiredImprovementRate` can not work as expected when `previousInterestRate` less than 10 due to precision loss
- [M-05] Borrowers lose funds if they call `repayAndCloseLoan` instead of `closeLoan`
- [M-06] Might not get desired min loan amount if `_originationFeeRate` changes
- [M-07] `mintBorrowTicketTo` can be a contract with no `onERC721Received` method, which may cause the BorrowTicket NFT to be frozen and put users' funds at risk
- Low Risk and Non-Critical Issues
 - L-01 Loans can be created and paid with non-existent/destroyed tokens
 - L-02 `_originationFeeRate` s of less than 1000 may charge no fees if amounts are small
 - L-03 A malicious owner can keep the fee rate at zero, but if a large value transfer enters the mempool, the owner can jack the rate up to the maximum
 - L-04 A malicious owner can set an effectively infinite improvement rate with `type(uint256).max` after he/she has entered into a loan to prevent others from buying them out
 - L-05 `tokenURI()` reverts for tokens that don't implement `IERC20Metadata`
 - L-06 `_safeMint()` should be used rather than `_mint()` wherever possible
 - L-07 `loanFacilitatorTransfer()` does not verify that the receiver is capable of handling an NFT
 - L-08 Missing checks for `address(0x0)` when assigning values to `address` state variables
 - N-01 `constant` s should be defined rather than using magic numbers
 - N-02 Typos

- [N-03 NatSpec is incomplete](#)
- [N-04 Event is missing `indexed` fields](#)
- [Gas Optimizations \(23\)](#)
 - [Foreword](#)
 - [G-01 Storage](#)
 - [G-02 Comparisons](#)
 - [G-03 Arithmetics](#)
 - [G-04 Visibility](#)
 - [G-05 Errors](#)
- [Disclosures](#)



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 Backed Protocol smart contract system written in Solidity. The audit contest took place between April 5—April 7 2022.



Wardens

54 Wardens contributed reports to the Backed Protocol contest:

1. [rayn](#)
2. OxDjango
3. [cmichel](#)
4. WatchPug ([jtp](#) and [ming](#))

5. hake
6. llllll
7. [teryanarmen](#)
8. [Dravee](#)
9. Certoralnc ([danb](#), egjlmn1, [OriDabush](#), ItayG, and shakedwinder)
10. [csanuragjain](#)
11. [hickuphh3](#)
12. [Ruhum](#)
13. [t11s](#)
14. tintin
15. joshie
16. robee
17. AuditsAreUS
18. [danb](#)
19. [berndartmueller](#)
20. [shenwilly](#)
21. [jah](#)
22. minhquanym
23. TerrierLover
24. cccz
25. saian
26. Oxkatana
27. reassor
28. [securerodd](#)
29. [Ov3rf10w](#)
30. sorrynotsorry
31. FSchmoede
32. Ox1f8b
33. [Kenshin](#)

34. [MetaOxNull](#)

35. [z3s](#)

36. m9800

37. [rfa](#)

38. [BouSalman](#)

39. VAD37

40. PPrieditis

41. Hawkeye (Oxwags and Oxmint)

42. horsefacts

43. hubble (ksk2345 and shri4net)

44. samruna

45. [Tomio](#)

46. [Funen](#)

47. [obront](#)

This contest was judged by [gzeon](#).

Final report assembled by [liveactionllama](#).



Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 34 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 23 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.



Scope

The code under review can be found within the [C4 Backed Protocol contest repository](#), and is composed of 7 smart contracts written in the Solidity

programming language and includes 434 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/non-critical.

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 (3)



[H-01] Can force borrower to pay huge interest

Submitted by cmichel, also found by AuditsAreUS, csanuragjain, danb, llllll, joshie, Ruhum, t11s, and tintin

[NFTLoanFacilitator.sol#L148](#)

The loan amount is used as a min loan amount. It can be matched as high as possible (realistically up to the collateral NFT's worth to remain in profit) and the borrower has to pay interest on the entire amount instead of just on the desired loan amount when the loan was created.



Proof of Concept

- User needs a 10k USDC loan, NFTs are illiquid and they only have a BAYC worth 350k\$. So buying another NFT worth roughly the desired 10k\$ is not feasible. They will put the entire 350k\$ BAYC as collateral for the 10k USDC loan.
- A lender matches the loan calling `lend` with 350k USDC.
- The borrower now has to pay interest on the entire 350k USDC even though they only wanted a 10k loan. Otherwise, they risk losing their collateral. Their effective rate on their 10k loan is 35x higher.



Recommended Mitigation Steps

The loan amount should not have min amount semantics. When someone wants to get a loan, they specify a certain amount they need, they don't want to receive and pay interest on more than that.

[wilsoncusack \(Backed Protocol\) disputed and commented:](#)

The ability to increase the loan amount is seen as a feature of the protocol, not a bug.

[gzeon \(judge\) decreased severity to Medium and commented:](#)

While a larger loan size is strictly beneficial to the borrower, the higher interest payment it entitled is not. The warden suggested a valid situation that may cost the user more than intended. Considering the amount lost is bounded because the lender carry more risk for a larger loan, downgrading this to Medium risk for the sponsor to consider a `maxLoanAmount` parameter.

[gzeon \(judge\) increased severity to High and commented:](#)

After considering [#9](#) bringing up the loan origination fee, I believe this is a High risk issue for the protocol to not have a `maxLoanAmount` parameter.

[wilsoncusack \(Backed Protocol\) commented:](#)

IMO it does not make sense to label this as High severity. This is not an exploit but is just the protocol working exactly as described in the README.

[gzeon \(judge\) commented:](#)

From README

Perpetual lender buyout: a lender can be boughtout at any time by a new lender who meets the existing terms and beats at least one term by at least 10%, e.g. 10% longer duration, 10% higher loan amount, 10% lower interest. The new lender pays the previous lender their principal + any interest owed. The loan duration restarts on buyout.

I don't agree that allowing higher loan amount necessarily means the loan amount needs to be unbounded. Given the increased interest and origination fee, a higher loan amount is not necessarily "beating existing terms" as described in the README.

[wilsoncusack \(Backed Protocol\) commented:](#)

It certainly doesn't necessarily mean that but it is how we chose to implement it and I think the description is clear that there is no cap. We define "beating" as having one of those values changed by at least 10% and so I think it is beating as described by the readme.

Nonetheless, I appreciate your drawing focus again to this point ([we discussed on twitter](#) with our community during audit as this became a point of interest, and have of course considered this idea when designing the protocol at the outset). We will again consider adding a Boolean flag to each loan as to whether the borrower allows loan amount increases

[wilsoncusack \(Backed Protocol\) commented:](#)

Respect judge to have final say, but since this is going public want to make sure our take on this is clear.

We believe the protocol design was clearly communicated in the README, including origination fee and the possibility for perpetually increasing loan amount. We think there is no "exploit" here, just people pointing out potential downsides to how the protocol is designed (as one might point out problems of impermanent loss with an AMM.) We view these as QA reports. We are interested in this feedback and listening to it in that we want to listen to potential users and make sure our protocol appeals to as many people as possible.

[gzeon \(judge\) commented:](#)

I consider this as an exploit because asset can be lost. Combining unbounded loan amount, interest rate and origination fee (max 5%), a malicious lender can grief borrower with limited risk and get a chance to seize the collateral as demonstrated in the POC.

The fact that the code is working as described in README is irrelevant if it is going to make user lose their asset. If this is going to stay as a protocol design, I recommend to clearly communicate the risk of unbounded loan amount which is lacking in the contest repo.

[wilsoncusack \(Backed Protocol\) resolved](#)



[H-02] `currentLoanOwner` can manipulate `loanInfo` when any lenders try to buyout

Submitted by rayn

[NFTLoanFacilitator.sol#L205-L208](#)

[NFTLoanFacilitator.sol#L215-L218](#)

If an attacker already calls `lend()` to lend to a loan, the attacker can manipulate `loanInfo` by reentrancy attack when any lenders try to buyout. The attacker can set bad values of `lendInfo` (e.g. very long duration, and 0 interest rate) that the lender who wants to buyout don't expect.



Proof of Concept

An attacker lends a loan, and `loanAssetContractAddress` in `loanInfo` is ERC777 which is suffering from reentrancy attack. When a lender (victim) try to buyout the loan of the attacker:

1. The victim called `lend()` .
2. In `lend()` , it always call `ERC20(loanAssetContractAddress).safeTransfer` to send `accumulatedInterest + previousLoanAmount` to `currentLoanOwner` (attacker).

3. The `transfer` of `loanAssetContractAddress` `ERC777` will call `_callTokensReceived` so that the attacker can call `lend()` again in reentrancy with parameters:

- `loanId`: same loan Id
- `interestRate`: set to bad value (e.g. 0)
- `amount`: same amount
- `durationSeconds`: set to bad value (e.g. a long `durationSeconds`)
- `sendLendTicketTo`: same address of the attacker (`currentLoanOwner`)

4. Now the variables in `loanInfo` are changed to bad value, and the victim will get the lend ticket but the loan term is manipulated, and can not set it back (because it requires a better term).



Tools Used

`vim`



Recommended Mitigation Steps

Use `nonReentrant` modifier on `lend()` to prevent reentrancy attack:

[OpenZeppelin/ReentrancyGuard.sol](#)

[wilsoncusack \(Backed Protocol\) acknowledged, but disagreed with High severity and commented:](#)

┃ We should mitigate, but will think on this.

[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

┃ Not sure whether this should be Medium or High risk.

[wilsoncusack \(Backed Protocol\) commented:](#)

┃ Thinking more, again we should definitely mitigate, but I think this is less severe because I do not think `ERC777` tokens will work with our contract? The on received call will revert? So this would need to be a malicious `ERC20` designed just for this.

[wilsoncusack \(Backed Protocol\) resolved and commented:](#)

ererc777 does work because reception ack is not needed in the normal case.

[gzeon \(judge\) commented:](#)

Sponsor confirmed with fix.



[H-03] Borrower can be their own lender and steal funds from buyout due to reentrancy

Submitted by OxDjango

[NFTLoanFacilitator.sol#L214-L221](#)

[NFTLoanFacilitator.sol#L230-L250](#)

If borrower lends their own loan, they can repay and close the loan before ownership of the lend ticket is transferred to the new lender. The borrower will keep the NFT + loan amount + accrued interest.



Proof of Concept

This exploit requires that the `loanAssetContractAddress` token transfers control to the receiver.



Steps of exploit:

- Borrower creates loan with `createLoan()`.
- The same Borrower calls `lend()`, funding their own loan. The Borrower receives the lend ticket, and funds are transferred to themselves.
- A new lender attempts to buy out the loan. The original loan amount + accruedInterest are sent to the original lender (same person as borrower).
- Due to lack of checks-effects-interactions pattern, the borrower is able to immediately call `repayAndCloseLoan()` before the lend ticket is transferred to the new lender.

The following code illustrates that the new lender sends funds to the original lender prior to receiving the lend ticket in return.

```
    } else {
        ERC20(loan.loanAssetContractAddress).safeTransferFrom(
            msg.sender,
            currentLoanOwner,
            accumulatedInterest + previousLoanAmount
        );
    }
    ILendTicket(lendTicketContract).loanFacilitatorTransfer(currentL
```

The original lender/borrower calls the following `repayAndCloseI

```
function repayAndCloseLoan(uint256 loanId) external override not
    Loan storage loan = loanInfo[loanId];
```

```
    uint256 interest = _interestOwed(
        loan.loanAmount,
        loan.lastAccumulatedTimestamp,
        loan.perAnumInterestRate,
        loan.accumulatedInterest
    );
    address lender = IERC721(lendTicketContract).ownerOf(loanId)
    loan.closed = true;
    ERC20(loan.loanAssetContractAddress).safeTransferFrom(msg.se
    IERC721(loan.collateralContractAddress).safeTransferFrom(
        address(this),
        IERC721(borrowTicketContract).ownerOf(loanId),
        loan.collateralTokenId
    );

    emit Repay(loanId, msg.sender, lender, interest, loan.loanAn
    emit Close(loanId);
}
```

Finally, the new lender receives the lend ticket that has no utility at this point. The borrower now possesses the NFT, original loan amount, and accrued interest.

Recommended Mitigation Steps

Move the line to transfer the lend ticket to the new lender above the line to transfer to funds to the original lender. Or, use reentrancyGuard from OpenZeppelin to remove the risk of reentrant calls completely.

If desired, also require that the lender cannot be the same account as the borrower of a loan.

[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

Borrower would need to convince lender to use an ERC20 with this malicious callback, but yes is legit.

malicious ERC20

-> transfers value to borrow ticket holder

-> calls repay and close loan (would need funds available to do so, but still nets up)

[wilsoncusack \(Backed Protocol\) commented:](#)

Possibility of an ERC777 loan asset warrants this as high, I think. Even though the warden didn't suggest that vector.

[wilsoncusack \(Backed Protocol\) commented:](#)

Scratch that, I think ERC777 not possible because our contract isn't setup to receive them.

[wilsoncusack \(Backed Protocol\) commented:](#)

erc777 does work because reception ack is not needed in the normal case.

[wilsoncusack \(Backed Protocol\) resolved](#)

[gzeon \(judge\) commented:](#)

Sponsor confirmed.



Medium Risk Findings (7)



[M-01] When an attacker lends to a loan, the attacker can trigger DoS that any lenders can not buyout it

Submitted by rayn, also found by hake

[NFTLoanFacilitator.sol#L205-L208](#)

[NFTLoanFacilitator.sol#L215-L218](#)

If an attacker (lender) lends to a loan, the attacker can always revert transactions when any lenders try to buyout, making anyone can not buyout the loan of the attacker.



Proof of Concept

1. A victim calls `lend()` , trying to buyout the loan of the attacker.
2. In `lend()` , it always call `ERC20(loanAssetContractAddress).safeTransfer` to send `accumulatedInterest + previousLoanAmount` to `currentLoanOwner` (attacker).
3. If the transfer of `loanAssetContractAddress` is ERC777, it will call `_callTokensReceived` that the attacker can manipulate and always revert it.
4. Because `NFTLoanFacilitator` uses `safeTransfer` and `safeTransferFrom` to check return value, the transaction of the victim will also be reverted. It makes anyone can not buyout the loan of the attacker.

In `_callTokensReceived` , the attacker just wants to revert the buyout transaction, but keep `repayAndCloseLoan` successful. The attacker can call `loanInfoStruct(uint256 loanId)` in `_callTokensReceived` to check if the value of `loanInfo` is changed or not to decide to revert it.



Tools Used

vim



Recommended Mitigation Steps

Don't transfer `ERC20(loanAssetContractAddress)` to `currentLoanOwner` in `lend()` , use a global mapping to record redemption of lenders and add an external function `redeem` for lenders to transfer `ERC20(loanAssetContractAddress)` .

[wilsoncusack \(Backed Protocol\) acknowledged, but disagreed with High severity and commented:](#)

I think this is just part of perils of working with certain assets and I am not sure we will mitigate.

[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

Sorry, I lost track of this one/labeled incorrectly. This is indeed an issue we intend to address: we will block `erc777` tokens.

The worse implication here is that a lender could prevent a borrower from repaying and could seize the NFT.

Still not sure if this should be High or Medium. But there are legit `ERC777` tokens that a borrower might selecting unknowingly, so probably is High?

[gzeon \(judge\) decreased severity to Medium and commented:](#)

I suggest this as Med Risk as no fund is loss by preventing buyout.

[wilsoncusack \(Backed Protocol\) commented:](#)

But as I said above the bigger issue is they could block repayment, guaranteeing default and seizure of collateral?

[gzeon \(judge\) commented:](#)

I think you are correct as there is a similar call in [L241](#). However, both wardens failed to describe such attack path and I am inclined to keep this as Med Risk.

[wilsoncusack \(Backed Protocol\) resolved](#)

[M-02] Protocol doesn't handle fee on transfer tokens

Submitted by OxDjango, also found by cccz, csanuragjain, Dravee, llllll, robee, and Ruhum

[NFTLoanFacilitator.sol#L155-L160](#)

Since the borrower is able to specify any asset token, it is possible that loans will be created with tokens that support fee on transfer. If a fee on transfer asset token is chosen, the protocol will contain a point of failure on the original `lend()` call.

It is my belief that this is a medium severity vulnerability due to its ability to impact core protocol functionality.



Proof of Concept

For the first lender to call `lend()`, if the transfer fee % of the asset token is larger than the origination fee %, the second transfer will fail in the following code:

```
ERC20 (loanAssetContractAddress).safeTransferFrom(msg.sender, address(
uint256 facilitatorTake = amount * originationFeeRate / SCALAR;
ERC20 (loanAssetContractAddress).safeTransfer(
    IERC721 (borrowTicketContract).ownerOf (loanId),
    amount - facilitatorTake
);
```

Example:

- `originationFee = 2%` **Max fee is 5% per comments**
- `feeOnTransfer = 3%`
- `amount = 100 tokens`
- **Lender transfers** `amount`
- NFTLoanFacilitator **receives** `97`.
- `facilitatorTake = 2`
- NFTLoanFacilitator **attempts to send** `100 - 2` to borrower, but only has `97`.

- Execution reverts.



Other considerations:

If the `originationFee` is less than or equal to the `transferFee`, the transfers will succeed but will be received at a loss for the borrower and lender. Specifically for the lender, it might be unwanted functionality for a lender to lend 100 and receive 97 following a successful repayment (excluding interest for this example).



Recommended Mitigation Steps

Since the `originationFee` is calculated based on the `amount` sent by the lender, this calculation will always underflow given the example above. Instead, a potential solution would be to calculate the `originationFee` based on the requested loan amount, allowing the lender to send a greater value so that `feeOnTransfer <= originationFee`.

Oppositely, the protocol can instead calculate the amount received from the initial transfer and use this amount to calculate the `originationFee`. The issue with this option is that the borrower will receive less than the desired loan amount.

[wilsoncusack \(Backed Protocol\) commented:](#)

If `amount - origination_fee - token_fee < 0`, then yeah you will not be able to underwrite to loan. But that would be a huge fee.

[wilsoncusack \(Backed Protocol\) commented:](#)

Discussed more with warden OxDjango, if the token even has a 1% fee then the second transfer will fail OR we will leak facilitator funds, although this is sort of impossible because currently none of the transactions with these fee on transfer tokens will work.

[wilsoncusack \(Backed Protocol\) acknowledged and commented:](#)

This issue is slightly different from others that just point out that borrowers will get `amount - token_fee`. The only one I have seen, I think, to point out that fulfilling loans with fee on transfer tokens is impossible.

Imagine a token that takes 1% on transfer.

Amount = 100

99 reaches facilitator

facilitator transfers 100 - facilitator take = 99 to the borrower.

Facilitator gets nothing

Borrower gets 98.

If the facilitator take is greater or the fee on transfer take is greater, it won't work at all.

Med severity is maybe right given we can miss out on origination fees?

[wilsoncusack \(Backed Protocol\) confirmed and resolved](#)

[gzeon \(judge\) commented:](#)

Sponsor confirmed with fix.



[M-03] `sendCollateralTo` is unchecked in `closeLoan()`, which can cause user's collateral NFT to be frozen

Submitted by WatchPug, also found by berndartmueller, Dravee, hake, jah, and minhquanyam

[NFTLoanFacilitator.sol#L116-L126](#)

```
function closeLoan(uint256 loanId, address sendCollateralTo) external
    require(IERC721(borrowTicketContract).ownerOf(loanId) == msgSender,
        "NFTLoanFacilitator: borrow ticket holder only");

    Loan storage loan = loanInfo[loanId];
    require(loan.lastAccumulatedTimestamp == 0, "NFTLoanFacilitator: loan not closed");

    loan.closed = true;
    IERC721(loan.collateralContractAddress).transferFrom(address(this), sendCollateralTo, loan.collateralId);
    emit Close(loanId);
}
```

The `sendCollateralTo` will receive the collateral NFT when `closeLoan()` is called. However, if `sendCollateralTo` is a contract address that does not support ERC721, the collateral NFT can be frozen in the contract.

As per the documentation of EIP-721:

A wallet/broker/auction application MUST implement the wallet interface if it will accept safe transfers.

Ref: [EIP-721](#)



Recommended Mitigation Steps

Change to:

```
function closeLoan(uint256 loanId, address sendCollateralTo) external
    require(IERC721(borrowTicketContract).ownerOf(loanId) == msg.sender,
        "NFTLoanFacilitator: borrow ticket holder only");

    Loan storage loan = loanInfo[loanId];
    require(loan.lastAccumulatedTimestamp == 0, "NFTLoanFacilitator: already closed");

    loan.closed = true;
    IERC721(loan.collateralContractAddress).safeTransferFrom(address(this), sendCollateralTo, loanId);
    emit Close(loanId);
}
```

[wilsoncusack \(Backed Protocol\) acknowledged, but disagreed with Medium severity and commented:](#)

We use `transferFrom` and `_mint` instead of the safe versions to save gas. We think it is a reasonable expectation that users calling this should know what they are doing. We feel this is OK especially because other major protocols like Uniswap do this ([Uniswap/NonfungiblePositionManager.sol#L156](#)).

[gzeon \(judge\) commented:](#)

I believe Med Risk is a fair assessment given the mixed/inconsistent usage of `safeTransferFrom` and `transferFrom` in the contract.



[M-04] `requiredImprovementRate` can not work as expected when `previousInterestRate` less than 10 due to precision loss

Submitted by WatchPug, also found by Certoralnc and hickuphh3

[NFTLoanFacilitator.sol#L167-L179](#)

```
{
    uint256 previousInterestRate = loan.perAnumInterestRate;
    uint256 previousDurationSeconds = loan.durationSeconds;

    require(interestRate <= previousInterestRate, 'NFTLoanFacilitator: interestRate must be less than or equal to previousInterestRate');
    require(durationSeconds >= previousDurationSeconds, 'NFTLoanFacilitator: durationSeconds must be greater than or equal to previousDurationSeconds');

    require((previousLoanAmount * requiredImprovementRate / SCAI
    || previousDurationSeconds + (previousDurationSeconds * requiredImprovementRate / SCAI
    || (previousInterestRate != 0 // do not allow rate improvement if previousInterestRate is 0
    && previousInterestRate - (previousInterestRate * requiredImprovementRate / SCAI) < 0)
    "NFTLoanFacilitator: proposed terms must be better than existing terms")
}
```

The `requiredImprovementRate` represents the percentage of improvement required of at least one of the terms when buying out from a previous lender.

However, when `previousInterestRate` is less than 10 and `requiredImprovementRate` is 100, due to precision loss, the new `interestRate` is allowed to be the same as the previous one.

Making such an expected constraint absent.



Proof of Concept

1. Alice `createLoan()` with `maxPerAnumInterest` = 10, received `loanId` = 1
2. Bob `lend()` with `interestRate` = 9 for `loanId` = 1
3. Charlie `lend()` with `interestRate` = 9 (and all the same other terms with Bob) and buys out `loanId` = 1

Charlie is expected to provide at least 10% better terms, but actually bought out Bob with the same terms.



Recommended Mitigation Steps

Consider using: [OpenZeppelin/Math.sol#L39-L42](#)

And change the check to:

```
(previousInterestRate != 0 // do not allow rate improvement if r
    && previousInterestRate - Math.ceilDiv(previousInterestF
```

[wilsoncusack \(Backed Protocol\) confirmed and resolved](#)

[gzeon \(judge\) commented:](#)



Sponsor confirmed with fix.



[M-05] Borrowers lose funds if they call `repayAndCloseLoan` instead of `closeLoan`

Submitted by cmichel

[NFTLoanFacilitator.sol#L241](#)

The `repayAndCloseLoan` function does not revert if there has not been a lender for a loan (matched with `lend`). Users should use `closeLoan` in this case but the contract should disallow calling `repayAndCloseLoan` because users can lose funds.

It performs a

`ERC20 (loan.loanAssetContractAddress).safeTransferFrom(msg.sender, lender, interest + loan.loanAmount)` call where `interest` will be a high value accumulated from timestamp 0 and the `loan.loanAmount` is the initially desired min loan amount `minLoanAmount` set in `createLoan`. The user will lose these funds if they ever approved the contract (for example, for another loan).



Recommended Mitigation Steps

Add a check that there actually is something to repay.

```
require(loan.lastAccumulatedTimestamp > 0, "loan was never match
```

[wilsoncusack \(Backed Protocol\) commented:](#)

ownerOf query here will fail if there is no lender, [NFTLoanFacilitator.sol#L239](#).

[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

Actually this is wrong, we switched to solmate and this ownerOf will not fail. Is a legit issue.

Not an attack, but funds can be lost some. Medium probably makes sense?

[wilsoncusack \(Backed Protocol\) commented:](#)

Requires borrow to have approved the facilitator to move this erc20 and to call the wrong method.

[wilsoncusack \(Backed Protocol\) resolved and commented:](#)

Yooo just discovered solmate had not followed the ERC721 standard on [this ownerOf](#) and it should have reverted, updated here

[Rari-Capital/solmate@921a9ad](#)

[gzeon \(judge\) commented:](#)

Sponsor confirmed with fix.



[M-06] Might not get desired min loan amount if
_originationFeeRate changes

Submitted by cmichel, also found by teryanarmen

Admins can update the origination fee by calling `updateOriginationFeeRate`. Note that a borrower does not receive their `minLoanAmount` set in `createLoan`, they only receive $(1 - \text{originationFee}) * \text{minLoanAmount}$, see [lend](#). Therefore, they need to precalculate the `minLoanAmount` using the **current** origination fee to arrive at the post-fee amount that they actually receive. If admins then increase the fee, the borrower receives fewer funds than required to cover their rent and might become homeless.



Recommended Mitigation Steps

Reconsider how the min loan amount works. Imo, this `minLoanAmount` should be the post-fee amount, not the pre-fee amount. It's also more intuitive for the borrower when creating the loan.

[wilsoncusack \(Backed Protocol\) disputed and commented:](#)

Won't change, is just how it works.

[wilsoncusack \(Backed Protocol\) acknowledged, but disagreed with Medium severity and commented:](#)

The only true mitigation here would be to store `originationFeeRate` in the `Loan` struct at the time of origination to guarantee a borrower gets the fee rate that was present when they created the loan. But we do not plan to make this change.

[wilsoncusack \(Backed Protocol\) resolved and commented:](#)

Decided to fix because we could do so without too much gas.

[gzeon \(judge\) commented:](#)

Sponsor confirmed with fix.



[M-07] `mintBorrowTicketTo` can be a contract with no

onERC721Received method, which may cause the BorrowTicket NFT to be frozen and put users' funds at risk

Submitted by WatchPug

[NFTLoanFacilitator.sol#L102-L102](#)

```
IERC721Mintable(borrowTicketContract).mint(mintBorrowTicketTo, i
```

[NFTLoanTicket.sol#L33-L35](#)

```
function mint(address to, uint256 tokenId) external override loa
    _mint(to, tokenId);
}
```

If `mintBorrowTicketTo` is a contract that does not implement the `onERC721Received` method, in the current implementation of `createLoan()`, the tx will still be successfully, and the loan will be created.

This can be a problem if `mintBorrowTicketTo` can not handle ERC721 properly, as the `BorrowTicket` NFT will be used later to get back the user's funds.



Recommended Mitigation Steps

Consider using `safeMint` in `NFTLoanTicket.sol#mint()`:

```
function mint(address to, uint256 tokenId) external override loa
    _safeMint(to, tokenId);
}
```

[wilsoncusack \(Backed Protocol\) acknowledged, but disagreed with Medium severity and commented:](#)

Similar to [#83](#)

[gzeon \(judge\) commented:](#)

Not really a duplicate because it is the mint function. Fund can be lost by losing the borrow ticket.

[wilsoncusack \(Backed Protocol\)](#) commented:

Agree not really duplicate. I think my response is the same, which is comparing to Uniswap v3 nft which also would mean loss of funds if lost.

From [#83](#):

We use transferFrom and _mint instead of the safe versions to save gas. We think it is a reasonable expectation that users calling this should know what they are doing, we feel this is OK especially because other major protocols like Uniswap do this [NonfungiblePositionManager.sol#L156](#).


[gzeon \(judge\)](#) commented:

Given the safe variant is used in [L242](#) and [L262](#), looks like ERC721 safety was a concern at the time the code is written. Therefore I believe Med Risk is a fair assessment.

[wilsoncusack \(Backed Protocol\)](#) commented:

Fair. This was an intentional change to save gas but I should have been consistent.

removed safeMint and safeTransfer

 master

 **wilsoncusack** committed 13 days ago



Low Risk and Non-Critical Issues

For this contest, 34 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](#) by warden [IIIIII](#) received the top score from the judge.

The following wardens also submitted reports: [Dravee](#), [shenwilly](#), [teryanarmen](#), [hake](#), [reassor](#), [TerrierLover](#), [OxDjango](#), [m9800](#), [robee](#), [securerodd](#), [t1ls](#), [tintin](#),

[Ov3rf10w](#), [Oxkatana](#), [berndartmueller](#), [BouSalman](#), [Certoralnc](#), [FSchmoede](#), [rayn](#), [Ruhum](#), [sorrynotsorry](#), [VAD37](#), [PPrieditis](#), [Ox1f8b](#), [csanuragjain](#), [Hawkeye](#), [horsefacts](#), [hubble](#), [Kenshin](#), [MetaOxNull](#), [samruna](#), [WatchPug](#), and [z3s](#).



[L-01] Loans can be created and paid with non-existent/destroyed tokens

`@rari-capital/solmate/src/utils/SafeTransferLib.sol` has functions named similarly to functions that OpenZeppelin has, but they act differently. At the top of the file is the following comment:

```
/// @dev Note that none of the functions in this library check t
```

[Rari-Capital/SafeTransferLib.sol#L9](#)

If the caller of these functions does not check that the token has code, calls to these functions will be no-ops, since low level calls to non-contracts always return success. There are many instances of these calls throughout the file with no code existence checks:

```
contracts/NFTLoanFacilitator.sol:155:         ERC20 (loanAsset
contracts/NFTLoanFacilitator.sol:157:         ERC20 (loanAsset
contracts/NFTLoanFacilitator.sol:200:             ERC20 (loanF
contracts/NFTLoanFacilitator.sol:205:             ERC20 (loanF
contracts/NFTLoanFacilitator.sol:210:             ERC20 (loanF
contracts/NFTLoanFacilitator.sol:215:             ERC20 (loan.
contracts/NFTLoanFacilitator.sol:241:         ERC20 (loan.loanAsse
contracts/NFTLoanFacilitator.sol:242:         IERC721 (loan.collat
contracts/NFTLoanFacilitator.sol:262:         IERC721 (loan.collat
contracts/NFTLoanFacilitator.sol:297:         ERC20 (asset) .safeTr
```



[L-02] originationFeeRate s of less than 1000 may charge no fees if amounts are small

File: `contracts/NFTLoanFacilitator.sol` (line [156](#))

```
uint256 facilitatorTake = amount * originationFeeRate / SCALAR;
```

Add a `require()` for `facilitatorTake` to be non-zero if `originationFeeRate` is non-zero, or state the fee logic for small amounts



[L-03] A malicious owner can keep the fee rate at zero, but if a large value transfer enters the mempool, the owner can jack the rate up to the maximum

File: `contracts/NFTLoanFacilitator.sol` (lines [306-312](#))

```
function updateOriginationFeeRate(uint32 _originationFeeRate) external {
    require(_originationFeeRate <= 5 * (10 ** (INTEREST_RATE_DECIMALS)));

    originationFeeRate = _originationFeeRate;

    emit UpdateOriginationFeeRate(_originationFeeRate);
}
```

Store the fee rate during loan creation, along with the maximum fee rate the user will allow, and update to the new rate for that particular loan only when loans are bought out



[L-04] A malicious owner can set an effectively infinite improvement rate with `type(uint256).max` after he/she has entered into a loan to prevent others from buying them out

File: `contracts/NFTLoanFacilitator.sol` (lines [320-326](#))

```
function updateRequiredImprovementRate(uint256 _improvementRate) external {
    require(_improvementRate > 0, 'NFTLoanFacilitator: 0 improve');

    requiredImprovementRate = _improvementRate;

    emit UpdateRequiredImprovementRate(_improvementRate);
}
```

Have a sane upper limit to the improvement rate, and don't allow it to change as above



[L-05] `tokenURI()` reverts for tokens that don't implement `IERC20Metadata`

While the ticket descriptors are not in scope, the code calling them is.

`NFTLoanTicket.tokenURI()`, which is in scope, ends up calling descriptor code which casts the asset to `IERC20Metadata`. This interface is separate from `IERC20` because EIP-20 does not require those functions to exist. If a valid ERC20 token does not implement this interface, casting it and attempting to call non-existent functions will cause the code to revert, which will cause `tokenURI()` to revert.

[PopulateSVGParams.sol#L65](#)

[PopulateSVGParams.sol#L69](#)

[PopulateSVGParams.sol#L83](#)

Use `safeDecimals()` etc



[L-06] `_safeMint()` should be used rather than `_mint()` wherever possible

`_mint()` is [discouraged](#) in favor of `_safeMint()` which ensures that the recipient is either an EOA or implements `IERC721Receiver`. Both open [OpenZeppelin](#) and [solmate](#) have versions of this function so that NFTs aren't lost if they're minted to contracts that cannot transfer them back out.

File: `contracts/NFTLoanTicket.sol` (line [34](#))

```
_mint(to, tokenId);
```



[L-07] `loanFacilitatorTransfer()` does not verify that the receiver is capable of handling an NFT

EIP-721 states:

```
/// @notice Transfer ownership of an NFT -- THE CALLER IS RESPONSIBLE
/// TO CONFIRM THAT `_to` IS CAPABLE OF RECEIVING NFTS OR ELSE
/// THEY MAY BE PERMANENTLY LOST
```

[EIP-721.md#L103-L105](#)

The code below was copied from `transferFrom()` , so any function calling `_transfer()` needs to confirm that `to` is capable of receiving NFTs.

`loanFacilitatorTransfer()` calls `_transfer()` without completing this check, which can lead to the loss of NFTs. Checking if the address is zero or not is not sufficient; it needs the other checks in [safeTransferFrom\(\)](#) .

File: contracts/LendTicket.sol (lines [24-34](#))

```
    /// @dev exact copy of
96
97  /// with L78 - L81 removed to enable loanFacilitatorTransfer
98  function _transfer(
99      address from,
100     address to,
101     uint256 id
102 ) internal {
103     require(from == ownerOf[id], "WRONG_FROM");
104
105     require(to != address(0), "INVALID_RECIPIENT");
```



[L-08] Missing checks for `address(0x0)` when assigning values to address state variables

1. File: contracts/NFTLoanFacilitator.sol (line [282](#))

```
lendTicketContract = _contract;
```

2. File: contracts/NFTLoanFacilitator.sol (line [292](#))

```
borrowTicketContract = _contract;
```



[N-01] constant s should be defined rather than using magic numbers

1. File: contracts/NFTLoanFacilitator.sol (line [307](#))

```
require(_originationFeeRate <= 5 * (10 ** (INTEREST_RATE_DECIMAL
```

2. File: contracts/NFTLoanFacilitator.sol (line [384](#))

```
* (perAnumInterestRate * 1e18 / 365 days)
```

3. File: contracts/NFTLoanFacilitator.sol (line [384](#))

```
* (perAnumInterestRate * 1e18 / 365 days)
```

4. File: contracts/NFTLoanFacilitator.sol (line [385](#))

```
/ 1e21 // SCALAR * 1e18
```



[N-02] Typos

1. File: contracts/NFTLoanFacilitator.sol (line [303](#))

```
* @notice Updates originationFeeRate the faciliator keeps of eac
```

faciliator

2. File: contracts/interfaces/INFTLoanFacilitator.sol (line [65](#))

```
* @param minLoanAmount minimum loan amount
```

mimumum



[N-03] NatSpec is incomplete

1. File: contracts/interfaces/INFTLoanFacilitator.sol (lines [286-288](#))

```
* @param loanId The loan id
*/
function totalOwed(uint256 loanId) view external returns (uint256)
```

Missing: @return

2. File: contracts/interfaces/INFTLoanFacilitator.sol (lines [292-294](#))

```
* @param loanId The loan id
*/
function interestOwed(uint256 loanId) view external returns (uint256)
```

Missing: @return

3. File: contracts/interfaces/INFTLoanFacilitator.sol (lines [298-300](#))

```
* @param loanId The loan id
*/
function loanEndSeconds(uint256 loanId) view external returns (uint256)
```

Missing: @return



[N-04] Event is missing indexed fields

Each event should use three indexed fields if there are three or more fields

1. File: contracts/interfaces/INFTLoanFacilitator.sol (lines [68-77](#))

```
event CreateLoan(  
    uint256 indexed id,  
    address indexed minter,  
    uint256 collateralTokenId,  
    address collateralContract,  
    uint256 maxInterestRate,  
    address loanAssetContract,  
    uint256 minLoanAmount,  
    uint256 minDurationSeconds  
);
```

2. File: contracts/interfaces/INFTLoanFacilitator.sol (lines [93-99](#))

```
event Lend(  
    uint256 indexed id,  
    address indexed lender,  
    uint256 interestRate,  
    uint256 loanAmount,  
    uint256 durationSeconds  
);
```

3. File: contracts/interfaces/INFTLoanFacilitator.sol (line [145](#))

```
event WithdrawOriginationFees(address asset, uint256 amount, adc
```

4. File: contracts/interfaces/INFTLoanFacilitator.sol (line [152](#))

```
event UpdateOriginationFeeRate(uint32 feeRate);
```

5. File: contracts/interfaces/INFTLoanFacilitator.sol (line [159](#))

```
event UpdateRequiredImprovementRate(uint256 improvementRate);
```


[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

High quality! Will work through all.

[wilsoncusack \(Backed Protocol\) resolved and commented:](#)

[L-01] Fixed, we now check code.length
[L-02] Won't fix
[L-03] Fixed, origination fee is frozen with loan terms
[L-04] Won't fix
[L-05] Won't fix
[L-06] Won't fix
[L-07] Won't fix
[L-08] Won't fix
[N-01] Won't fix, too much gas
[N-02] Fixed
[N-03] Fixed
[N-04] Won't fix



Gas Optimizations (23)

For this contest, 23 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by warden Dravee received the top score from the judge.

The following wardens also submitted reports: [lllllll](#), [saian](#), [robee](#), [TerrierLover](#), [joshie](#), [rfa](#), [Oxkatana](#), [Ov3rf10w](#), [sorrynotsorry](#), [Tomio](#), [t1ls](#), [FSchmoede](#), [Certoralnc](#), [Funen](#), [rayn](#), [Ox1f8b](#), [csanuragjain](#), [Kenshin](#), [MetaOxNull](#), [obront](#), [securerodd](#), and [z3s](#).



Foreword

- `@audit` tags

The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.



[G-01] Storage



Caching storage values in memory

The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). Here, storage values should get cached in memory (see the `@audit` tags for further details):

```
contracts/NFTLoanFacilitator.sol:
174:             require((previousLoanAmount * requiredImp
175:             || previousDurationSeconds + (previousDur
176:             || (previousInterestRate != 0 // do not a
177:             && previousInterestRate - (previousIr

231         Loan storage loan = loanInfo[loanId];
232
233         uint256 interest = _interestOwed(
234:             loan.loanAmount, //@audit gas: should cache l
235             loan.lastAccumulatedTimestamp,
236             loan.perAnumInterestRate,
237             loan.accumulatedInterest
238         );
239         address lender = IERC721(lendTicketContract).owne
240         loan.closed = true;
241:         ERC20(loan.loanAssetContractAddress).safeTransfer
242         IERC721(loan.collateralContractAddress).safeTrans
243             address(this),
244             IERC721(borrowTicketContract).ownerOf(loanId)
245             loan.collateralTokenId
246         );
247
248:         emit Repay(loanId, msg.sender, lender, interest,
249         emit Close(loanId);
250     }

338         Loan storage loan = loanInfo[loanId];
339         if (loan.closed || loan.lastAccumulatedTimestamp
340
341:         return loanInfo[loanId].loanAmount + _interestOwe
342             loan.loanAmount,
343             loan.lastAccumulatedTimestamp,
344             loan.perAnumInterestRate,
345             loan.accumulatedInterest
```



[G-02] Comparisons



`> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a `require` statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see [this tweet](#) for more proofs.

I suggest changing `> 0` with `!= 0` here:

```
NFTLoanFacilitator.sol:321:         require(_improvementRate > 0,
```

Also, please enable the Optimizer.



[G-03] Arithmetics



`++i` costs less gas compared to `i++` or `i += 1`

`++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning 1 instead of 2

Instances include:

```
LendTicket.sol:39:         balanceOf[from]--;
LendTicket.sol:41:         balanceOf[to]++;
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable. Same thing for `--i` and `i--`



Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block: [Checked or Unchecked Arithmetic](#).

I suggest wrapping with an `unchecked` block here (see `@audit` tags for more details):

```
contracts/NFTLoanFacilitator.sol:
156         uint256 facilitatorTake = amount * originationF
157         ERC20(loanAssetContractAddress).safeTransfer(
158             IERC721(borrowTicketContract).ownerOf(loanI
159:         amount - facilitatorTake //@audit gas: shou

209         uint256 facilitatorTake = (amountIncrease *
210         ERC20(loanAssetContractAddress).safeTransfe
211             IERC721(borrowTicketContract).ownerOf(I
212:         amountIncrease - facilitatorTake //@auc
```





Consider making some constants as non-public to save gas

Reducing from `public` to `private` or `internal` can save gas when a constant isn't used outside of its contract. I suggest changing the visibility from `public` to `internal` or `private` here:

```
NFTLoanFacilitator.sol:21:    uint8 public constant override INT
NFTLoanFacilitator.sol:24:    uint256 public constant override S
```



[G-05] Errors



Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
NFTLoanFacilitator.sol:118:    "NFTLoanFacilitator: borrow t
NFTLoanFacilitator.sol:121:    require(loan.lastAccumulatedI
NFTLoanFacilitator.sol:178:    "NFTLoanFacilitator:
NFTLoanFacilitator.sol:189:    "NFTLoanFacilitator: accu
NFTLoanFacilitator.sol:255:    "NFTLoanFacilitator: lend tic
NFTLoanFacilitator.sol:259:    "NFTLoanFacilitator: payment
NFTLoanTicket.sol:15:    require(msg.sender == address(nftLc
```

I suggest shortening the revert strings to fit in 32 bytes, or that using custom errors as described next.



Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source [Custom Errors in Solidity](#):

Starting from [Solidity v0.8.4](#), there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
LendTicket.sol:32:         require(from == ownerOf[id], "WRONG_FF
LendTicket.sol:34:         require(to != address(0), "INVALID_REC
NFTLoanFacilitator.sol:53:     require(!loanInfo[loanId].clos
NFTLoanFacilitator.sol:81:     require(minDurationSeconds !=
NFTLoanFacilitator.sol:82:     require(minLoanAmount != 0, 'N
NFTLoanFacilitator.sol:83:     require(collateralContractAddr
NFTLoanFacilitator.sol:85:     require(collateralContractAddr
NFTLoanFacilitator.sol:117:     require(IERC721(borrowTicketC
NFTLoanFacilitator.sol:121:     require(loan.lastAccumulatedI
NFTLoanFacilitator.sol:144:         require(loanAssetContract
NFTLoanFacilitator.sol:146:         require(interestRate <= 1
NFTLoanFacilitator.sol:147:         require(durationSeconds >
NFTLoanFacilitator.sol:148:         require(amount >= loan.lc
NFTLoanFacilitator.sol:171:         require(interestRate
NFTLoanFacilitator.sol:172:         require(durationSecor
NFTLoanFacilitator.sol:174:         require((previousLoar
NFTLoanFacilitator.sol:188:         require(accumulatedIntere
NFTLoanFacilitator.sol:254:     require(IERC721(lendTicketCor
NFTLoanFacilitator.sol:258:     require(block.timestamp > loa
NFTLoanFacilitator.sol:280:     require(lendTicketContract ==
NFTLoanFacilitator.sol:290:     require(borrowTicketContract
NFTLoanFacilitator.sol:307:     require(_originationFeeRate <
NFTLoanFacilitator.sol:321:     require(_improvementRate > 0,
NFTLoanTicket.sol:15:         require(msg.sender == address(nftLc
```

I suggest replacing revert strings with custom errors.

[wilsoncusack \(Backed Protocol\) confirmed and commented:](#)

Good stuff.

[wilsoncusack \(Backed Protocol\) resolved and commented:](#)

Fixed `loan.loanAmount` .

Fixed the error message size.

I don't think will change any others of these.

[wilsoncusack \(Backed Protocol\) commented:](#)

[G-01] Resolved
[G-02] Resolved
[G-03] Won't fix
[G-04] Won't fix
[G-05] Resolved



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