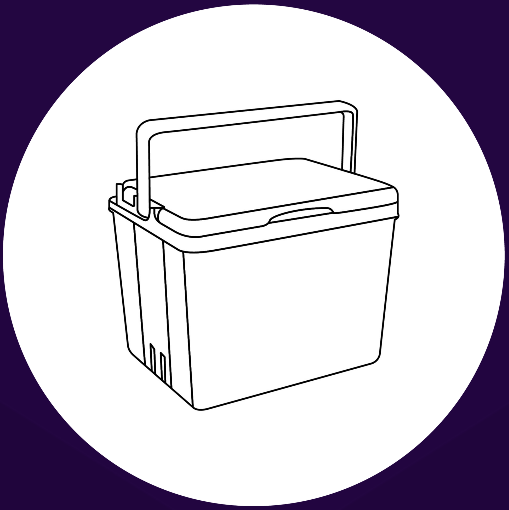
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:	cooler
Prepared by:	Sherlock
Lead Security Expert:	IIIIIII
Dates Audited:	January 20 - January 23, 2023
Prepared on:	February 13, 2023

Introduction

Cooler is a peer-to-peer lending protocol allowing a borrower and lender to engage in fixed-duration, fixed-interest lending. Cooler Loans are lightweight, trustless, independent of price-based liquidation.

Scope

Cooler.sol & Factory.sol ClearingHouse.sol

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
5	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

IIIIIII
libratus
0x52
wagmi
hansfriese
cccz
Allarious
HonorLt
HollaDieWaldfee
berndartmueller
Trumpero
Bahurum
Avci
serial-coder
stent
ElKu
ak1
kiki_dev
csanuragjain
simon135
zaskoh
bin2chen
Zarf
TrungOre
dipp
Breeje
yixxas

SHERLOCK

enckrish	Tricko	Madalad
usmannk	Nyx	8olidity
Deivitto	Metadev	0x4non
tsvetanovv	ahmedovv	0xSmartContract
ck	John	zaevlad
rvierdiiev	polthedev	Ityu
jonatascm	0xadrii	seyni
Atarpara	eyexploit	yongkiws
thekmj	MohanVarma	0xAgro
neumo	0xhacksmithh	ctrlc03
oxcm	supernova	psy4n0n
banditx0x	imare	sach1r0
cducrest-brainbot	ch0bu	gjaldon
ali_shehab	Qeew	peanuts

Issue H-1: Use safeTransfer/safeTransferFrom consistently instead of transfer/transferFrom

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/335

Found by

tsvetanovv, 0x52, polthedev, wagmi, enckrish, ak1, IllIIll, yongkiws, ctrlc03, zaskoh, Trumpero, TrungOre, Breeje, imare, jonatascm, cccz, Metadev, Nyx, neumo, Atarpara, serial-coder, yixxas, Tricko, 8olidity, Qeew, ahmedovv, libratus, usmannk, MohanVarma, psy4n0n, 0x4non, kiki_dev, peanuts, 0xhacksmithh, eyexploit, 0xSmartContract, supernova, Zarf, thekmj, Ityu, ck, sach1r0, hansfriese, John, HollaDieWaldfee, HonorLt, rvierdiiev, zaevlad, 0xAgro, Avci, gjaldon, Madalad, ch0bu, bin2chen, Bahurum, seyni, 0xadrii, Deivitto

Summary

Use safeTransfer/safeTransferFrom consistently instead of transfer/transferFrom

Vulnerability Detail

Some tokens do not revert on failure, but instead return false (e.g. ZRX). https://github.com/d-xo/weird-erc20/#no-revert-on-failure tranfser/transferfrom is directly used to send tokens in many places in the contract and the return value is not checked. If the token send fails, it will cause a lot of serious problems. For example, in the clear function, if debt token is ZRX, the lender can clear request without providing any debt token.

```
function clear (uint256 reqID) external returns (uint256 loanID) {
    Request storage req = requests[reqID];

    factory.newEvent(reqID, CoolerFactory.Events.Clear);

    if (!req.active)
        revert Deactivated();
    else req.active = false;

    uint256 interest = interestFor(req.amount, req.interest, req.duration);
    uint256 collat = collateralFor(req.amount, req.loanToCollateral);
    uint256 expiration = block.timestamp + req.duration;

    loanID = loans.length;
    loans.push(
        Loan(req, req.amount + interest, collat, expiration, true, msg.sender)
    );
    debt.transferFrom(msg.sender, owner, req.amount);
}
```

Impact

If the token send fails, it will cause a lot of serious problems. For example, in the clear function, if debt token is ZRX, the lender can clear request without providing any debt token.

Code Snippet

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L85-L86 https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L122-L123 https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L146-L147 https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L179-L180 https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L205-L206 https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L102-L103

Tool used

Manual Review

Recommendation

Consider using safeTransfer/safeTransferFrom consistently.

Discussion

hrishibhat

Sponsor comment:

> Good spot. Niche case.

Issue H-2: Loans can be rolled an unlimited number of times

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/215

Found by

0x52, enckrish, lllllll, cducrest-brainbot, banditx0x, simon135, Allarious, Trumpero, Breeje, neumo, Atarpara, yixxas, libratus, usmannk, ali_shehab, oxcm, thekmj, HollaDieWaldfee, HonorLt, bin2chen

Summary

Loans can be rolled an unlimited number of times, without letting the lender decide if has been done too many times already

Vulnerability Detail

The lender is expected to be able to toggle whether a loan can be rolled or not, but once it's enabled, there is no way to prevent the borrower from rolling an unlimited number of times in the same transaction or in quick succession.

Impact

If the lender is giving an interest-free loan and assumes that allowing a roll will only extend the term by one, they'll potentially be forced to wait until the end of the universe if the borrower chooses to roll an excessive number of times.

If the borrower is using a quickly-depreciating collateral, the lender may be happy to allow one a one-term extension, but will lose money if the term is rolled multiple times and the borrower defaults thereafter.

The initial value of `loan.rollable` is always `true`, so unless the lender calls `toggleRoll()` in the same transaction that they call `clear()`, a determined attacker will be able to roll as many times as they wish.

Code Snippet

As long as the borrower is willing to pay the interest up front, they can call `roll()` any number of times, extending the duration of the total loan to however long they wish:

```
// File: src/Cooler.sol : Cooler.roll()   #1

    function roll (uint256 loanID) external {
        Loan storage loan = loans[loanID];
        Request memory req = loan.request;

        if (block.timestamp > loan.expiry)
            revert Default();

        if (!loan.rollable)
            revert NotRollable();

        uint256 newCollateral = collateralFor(loan.amount, req.loanToCollateral) - loan.collateral;
        uint256 newDebt = interestFor(loan.amount, req.interest, req.duration);

        loan.amount += newDebt;
        loan.expiry += req.duration;
        loan.collateral += newCollateral;

        collateral.transferFrom(msg.sender, address(this), newCollateral);
    }
```

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L129-L147

toggleRoll() can't be used to stop rolls if they're all done in a single transaction.

Tool used

Manual Review

Recommendation

Have a variable controlling the number of rolls the lender is allowing, and or only allow a roll if the current `block.timestamp` is within one `req.duration` of the current `loan.expiry`

Discussion

hrishibhat

Sponsor comment:

> Will resolve as result of change for #265

Issue H-3: Fully repaying a loan will result in debt payment being lost

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/33

Found by

0x52, wagmi, serial-coder, HonorLt, stent, Avci, libratus, Bahurum, ElKu, berndartmueller

Summary

When a `loan` is fully repaid the `loan` storage is deleted. Since `loan` is a `storage` reference to the loan, `loan.lender` will return `address(0)` after the `loan` has been deleted. This will result in the `debt` being transferred to `address(0)` instead of the lender. Some ERC20 tokens will revert when being sent to `address(0)` but a large number will simply be sent there and lost forever.

Vulnerability Detail

```
function repay (uint256 loanID, uint256 repaid) external {
    Loan storage loan = loans[loanID];

    if (block.timestamp > loan.expiry)
        revert Default();

    uint256 decollateralized = loan.collateral * repaid / loan.amount;

    if (repaid == loan.amount) delete loans[loanID];
    else {
        loan.amount -= repaid;
        loan.collateral -= decollateralized;
    }

    debt.transferFrom(msg.sender, loan.lender, repaid);
    collateral.transfer(owner, decollateralized);
}
```

In `Cooler#repay` the loan storage associated with the loanID being repaid is deleted. `loan` is a storage reference so when `loans[loanID]` is deleted so is `loan`. The result is that `loan.lender` is now `address(0)` and the loan payment will be sent there instead.

Impact

Lender's funds are sent to `address(0)`

Code Snippet

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L108-L124

Tool used

Manual Review

Recommendation

Send collateral/debt then delete:

```
-   if (repaid == loan.amount) delete loans[loanID];
+   if (repaid == loan.amount) {
+       debt.transferFrom(msg.sender, loan.lender, loan.amount);
+       collateral.transfer(owner, loan.collateral);
+       delete loans[loanID];
+       return;
+   }
```

Discussion

hrishibhat

Sponsor comment:

> Great spot, embarassing oversight.

Issue H-4: Lender force Loan become default

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/23

Found by

hansfriese, 0x52, wagmi, IllIIII, bin2chen, Zarf, dipp, libratus, simon135, Trumpero, zaskoh, TrungOre, cccz

Summary

in `repay()` directly transfer the debt token to Lender, but did not consider that Lender can not accept the token (in contract blacklist), resulting in repay() always revert, and finally the Loan can only expire, Loan be default

Vulnerability Detail

The only way for the borrower to get the collateral token back is to repay the amount owed via repay(). Currently in the repay() method transfers the debt token directly to the Lender. This has a problem: if the Lender is blacklisted by the debt token now, the debtToken.transferFrom() method will fail and the repay() method will always fail and finally the Loan will default. Example: Assume collateral token = ETH,debt token = USDC, owner = alice 1.alice call request() to loan 2000 usdc , duration = 1 mon 2.bob call clear(): loanID =1 3.bob transfer loan[1].lender = jack by Cooler.approve/transfer
Note: jack has been in USDC's blacklist for some reason before or bob in USDC's blacklist for some reason now, it doesn't need transfer 'lender') 4.Sometime before the expiration date, alice call repay(id=1) , it will always revert, Because usdc.transfer(jack) will revert 5.after 1 mon, loan[1] default, jack call defaulted() get collateral token

```
    function repay (uint256 loanID, uint256 repaid) external {
        Loan storage loan = loans[loanID];
...
        debt.transferFrom(msg.sender, loan.lender, repaid);    //***<-------
↪   lender in debt token's blocklist will revert , example :debt = usdc
        collateral.transfer(owner, decollateralized);
    }
```

Impact

Lender forced Loan become default for get collateral token, owner lost collateral token

Code Snippet

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L122

Tool used

Manual Review

Recommendation

Instead of transferring the debt token directly, put the debt token into the Cooler.sol and set like: withdrawBalance[lender]+=amount, and provide the method withdraw() for lender to get debtToken back

Discussion

hrishibhat

Sponsor comment:

> Niche case + lender can transfer lender role to different, non-blacklisted wallet if needed.

IIIIIII000

The attacker in this case is the lender, so they wouldn't transfer to another wallet

hrishibhat

Agree with Lead Watson as the lender themself is the attacker here

Issue M-1: `Cooler.roll()` wouldn't work as expected when `newCollateral = 0`.

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/320

Found by

hansfriese, cccz, Allarious, csanuragjain

Summary

`Cooler.roll()` is used to increase the loan duration by transferring the additional collateral.

But there will be some problems when `newCollateral = 0`.

Vulnerability Detail

```
function roll (uint256 loanID) external {
    Loan storage loan = loans[loanID];
    Request memory req = loan.request;

    if (block.timestamp > loan.expiry)
        revert Default();

    if (!loan.rollable)
        revert NotRollable();

    uint256 newCollateral = collateralFor(loan.amount, req.loanToCollateral) -
    ↪ loan.collateral;
    uint256 newDebt = interestFor(loan.amount, req.interest, req.duration);

    loan.amount += newDebt;
    loan.expiry += req.duration;
    loan.collateral += newCollateral;

    collateral.transferFrom(msg.sender, address(this), newCollateral); //@audit
    ↪ 0 amount
}
```

In `roll()`, it transfers the `newCollateral` amount of collateral to the contract.

After the borrower repaid most of the debts, `loan.amount` might be very small and `newCollateral` for the original interest might be 0 because of the rounding issue.

Then as we can see from this one, some tokens might revert for 0 amount and `roll()` wouldn't work as expected.

Impact

There will be 2 impacts.

1. When the borrower tries to extend the loan using `roll()`, it will revert with the weird tokens when `newCollateral = 0`.
2. After the borrower noticed he couldn't repay anymore(so the lender will default the loan), the borrower can call `roll()` again when `newCollateral = 0`. In this case, the borrower doesn't lose anything but the lender must wait for `req.duration` again to default the loan.

Code Snippet

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L146

Tool used

Manual Review

Recommendation

I think we should handle it differently when `newCollateral = 0`.

According to impact 2, I think it would be good to revert when `newCollateral = 0`.

Discussion

hrishibhat

Sponsor comment:

> Good spot. Niche case.

Issue M-2: Loan is rollable by default

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/265

Found by

hansfriese, Nyx, enckrish, wagmi, yixxas, HollaDieWaldfee, HonorLt, Tricko, Zarf, libratus, simon135, usmannk, Trumpero

Summary

Making the loan rollable by default gives an unfair early advantage to the borrowers.

Vulnerability Detail

When clearing a new loan, the flag of `rollable` is set to true by default:

```
loans.push(
    Loan(req, req.amount + interest, collat, expiration, true, msg.sender)
);
```

This means a borrower can extend the loan anytime before the expiry:

```
function roll (uint256 loanID) external {
    Loan storage loan = loans[loanID];
    Request memory req = loan.request;

    if (block.timestamp > loan.expiry)
        revert Default();

    if (!loan.rollable)
        revert NotRollable();
```

If the lenders do not intend to allow rollable loans, they should separately toggle the status to prevent that:

```
function toggleRoll(uint256 loanID) external returns (bool) {
    ...
    loan.rollable = !loan.rollable;
    ...
}
```

I believe it gives an unfair advantage to the borrower because they can re-roll the loan before the lender's transaction forbids this action.

Impact

Lenders who do not want the loans to be used more than once, have to bundle their transactions. Otherwise, it is possible that someone might roll their loan, especially if the capital requirements are not huge because anyone can roll any loan.

Code Snippet

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L177

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L191

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L126-L147

Tool used

Manual Review

Recommendation

I believe `rollable` should be set to false by default or at least add an extra function parameter to determine the initial value of this status.

Discussion

hrishibhat

Sponsor comment:

> Valid. Will default to false.

sherlock-admin

> Retracted since https://github.com/sherlock-audit/2023-01-cooler-judging/issues/215 shows that there can be circumstances where funds lose value over the life of the loan

You've deleted an escalation for this issue.

Issue M-3: Repaying loans with small amounts of debt tokens can lead to underflowing in the `roll` function

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/263

Found by

tsvetanovv, rvierdiiev, ck, zaskoh, Allarious, Trumpero, Breeje, berndartmueller, jonatascm, Deivitto

Summary

Due to precision issues when repaying a loan with small amounts of debt tokens, the `loan.amount` can be reduced whereas the `loan.collateral` remains unchanged. This can lead to underflowing in the `roll` function.

Vulnerability Detail

The `decollateralized` calculation in the `repay` function rounds down to zero if the `repaid` amount is small enough. This allows iteratively repaying a loan with very small amounts of debt tokens without reducing the collateral.

The consequence is that the `roll` function can revert due to underflowing the `newCollateral` calculation once the `loan.collateral` is greater than `collateralFor(loan.amount, req.loanToCollateral)` (`loan.amount` is reduced by repaying the loan)

As any ERC-20 tokens with different decimals can be used, this precision issue is amplified if the decimals of the collateral and debt tokens differ greatly.

Impact

The `roll` function can revert due to underflowing the `newCollateral` calculation if the `repay` function is (iteratively) called with small amounts of debt tokens.

Code Snippet

Cooler.sol#L114

```
function repay (uint256 loanID, uint256 repaid) external {
    Loan storage loan = loans[loanID];

    if (block.timestamp > loan.expiry)
        revert Default();
    uint256 decollateralized = loan.collateral * repaid / loan.amount; //
↪   @audit-info (10e18 * 10) / 1_000e18 = 0 (rounds down due to imprecision)

    if (repaid == loan.amount) delete loans[loanID];
    else {
        loan.amount -= repaid;
        loan.collateral -= decollateralized;
    }

    debt.transferFrom(msg.sender, loan.lender, repaid);
    collateral.transfer(owner, decollateralized);
}
```

Cooler.sol#L139

Calculating `newCollateral` in L139 can potentially revert due to underflowing if `loan.collateral` is greater than the required collateral (`collateralFor(loan.amount, req.loanToCollateral)`).

A malicious user can use the imprecision issue in the `repay` function in L114 to repay small amounts of debt tokens (`loan.collateral * repaid` < `loan.amount`), which leads to no reduction of loan collateral, whereas the `loan.amount` is reduced.

This will prevent the `roll` function from being called.

```
function roll (uint256 loanID) external {
    Loan storage loan = loans[loanID];
    Request memory req = loan.request;

    if (block.timestamp > loan.expiry)
        revert Default();

    if (!loan.rollable)
        revert NotRollable();

    uint256 newCollateral = collateralFor(loan.amount, req.loanToCollateral) -
↪   loan.collateral;
    uint256 newDebt = interestFor(loan.amount, req.interest, req.duration);

    loan.amount += newDebt;
    loan.expiry += req.duration;
    loan.collateral += newCollateral;

    collateral.transferFrom(msg.sender, address(this), newCollateral);
}
```

Tool used

Manual Review

Recommendation

Consider preventing the loan from being repaid if the amount of returned collateral tokens is zero (i.e., `decollateralized == 0`).

Discussion

hrishibhat

Sponsor comment:

> Good Spot.

Issue M-4: Dust amounts can cause payments to fail, leading to default

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/218

Found by

kiki_dev, HollaDieWaldfee, IllIIII, ak1

Summary

Dust amounts can cause payments to fail, leading to default

Vulnerability Detail

In order for a loan to close, the exact right number of wei of the debt token must be sent to match the remaining loan amount. If more is sent, the balance underflows, reverting the transaction.

Impact

An attacker can send dust amounts right before a loan is due, front-running any payments also destined for the final block before default. If the attacker's transaction goes in first, the borrower will be unable to pay back the loan before default, and will lose thier remaining collateral. This may be the whole loan amount.

Code Snippet

If the repayment amount isn't exactly the remaining loan amount, and instead is more (due to the dust payment), the subtraction marked below will underflow, reverting the payment:

```
// File: src/Cooler.sol : Cooler.repay()   #1

       function repay (uint256 loanID, uint256 repaid) external {
           Loan storage loan = loans[loanID];

           if (block.timestamp > loan.expiry)
               revert Default();

           uint256 decollateralized = loan.collateral * repaid / loan.amount;

          if (repaid == loan.amount) delete loans[loanID];
          else {
@>             loan.amount -= repaid;
```

SHERLOCK

```
            loan.collateral -= decollateralized;
        }

        debt.transferFrom(msg.sender, loan.lender, repaid);
        collateral.transfer(owner, decollateralized);
    }
```

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L108-L124

Tool used

Manual Review

Recommendation

Only collect and subtract the minimum of the current loan balance, and the amount specified in the `repaid` variable

Discussion

hrishibhat

Sponsor comment:

> Good spot. Niche case.

Issue M-5: DAI/gOHM exchange rate may be stale

Source: https://github.com/sherlock-audit/2023-01-cooler-judging/issues/217

Found by

IllIllI

Summary

The `maxLTC` variable is a constant which implies a specific DAI/gOHM echange rate. The exchange rate has already changed so the current value in use will be wrong, and any value chosen now will eventually be out of date.

Vulnerability Detail

The `ClearingHouse` allows any loan to go through (assuming the `operator` approves it, and the `operator` is likely some sort of keeper program), and decides whether the terms are fair based on the hard-coded `maxLTC`, which will be (and is already - gOHM is currently worth $2,600) out of date.

If the code had been using a Chainlink oracle, this issue would be equivalent to not checking whether the price used to determine the loan-to-collateral ratio was stale, which is a Medium-severity issue.

It's not clear who or what exactly will be in control of the `operator` address which will make the `clear()` calls, but it will likely be a keeper which, unless programmed otherwise, would blindly approve such loans. Even if the `operator` is an actual person, the fact that there are coded checks for the `maxLTC`, means that the person/keeper can't be fully trusted, or that the code is attempting to protect against mistakes, so this category of mistake should also be added.

Impact

Under-collateralized loans will be given, and borrowers will purposely take loans default, since they can use the loan amount to buy more collateral than they would lose during default.

Code Snippet

The maximum loan-to-collateral is hard-coded, rather than being based on an oracle price:

```
// File: src/aux/ClearingHouse.sol : ClearingHouse.maxLTC   #1

34:@>     uint256 public constant maxLTC = 2_500 * 1e18; // 2,500
```

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/aux/ClearingHouse.sol#L34

If the gOHM price drops below $2500 to say $2000, a loan for 2500 DAI will only require 1 gOHM of collateral, even though it should require at least 1.2 gOHM in order to be fully-collateralized:

```
// File: src/Cooler.sol : Cooler.collateralFor()   #2

236        function collateralFor(uint256 amount, uint256 loanToCollateral)
 ↪  public pure returns (uint256) {
237 @>         return amount * decimals / loanToCollateral;
238:       }
```

https://github.com/sherlock-audit/2023-01-cooler/blob/main/src/Cooler.sol#L236-L238

Tool used

Manual Review

Recommendation

Use a chainlink oracle to determine the right prices to use when coming up with the maximum loan-to-collateral, for *each* loan

Discussion

hrishibhat

Sponsor comment:

> Not intended to be updated in real time. Set via gov; computed relative to backing (far less volatile than pricing).

IllIllI000

The values are stored in `constant` variables, so once they're set they cannot change

hrishibhat

Considering this issue as a valid medium, as a request placed with an ltc based on the exchange rate would result in incorrect maxltc calculated. Could render the contract useless since the maxLTC is fixed.

> If the gOHM price drops below $2500 to say $2000, a loan for 2500 DAI will only require 1 gOHM of collateral, even though it should require at least 1.2 gOHM in order to be fully-collateralized:

0x00052

Escalate for 11 USDC

`If the gOHM price drops below $2500 to say $2000, a loan for 2500 DAI will only require 1 gOHM of collateral, even though it should require at least 1.2 gOHM in order to be fully-collateralized:`

I disagree with this statement. It doesn't use the maxLTC when creating the loan, it uses the value specified by the loan request. maxLTC is just the max value it can't exceed. There's nothing forcing the operator to accept every loan at/under the maxLTC. The operator would have discretion and wouldn't clear bad loans that are undercollateralized from the start. The only thing it's doing it providing a hard stop for the ltc.

```
(
    uint256 amount,
    uint256 interest,
    uint256 ltc,
    uint256 duration,
) = cooler.requests(id);


// Validate terms
if (interest < minimumInterest)
    revert InterestMinimum();
if (ltc > maxLTC)
    revert LTCMaximum();
```

We see in the above lines that it pulls the requested LTC from the cooler and compares it to maxLTC to confirm it isn't too high. It doesn't use the maxLTC value, it simply functions as a guardrail to make sure it doesn't accidentally clear a bad loan to gives way too much DAI. Technically if the price of gOHM was to increase dramatically the contract would become useless, but it wouldn't lead to any lost/stuck funds and it would be easy enough to just deploy a new contract with different bounds.

This should be low severity.

sherlock-admin

> Escalate for 11 USDC
>
> `If the gOHM price drops below $2500 to say $2000, a loan for 2500 DAI will only require 1 gOHM of collateral, even though it should require at least 1.2 gOHM in order to be fully-collateralized:`
>
> I disagree with this statement. It doesn't use the maxLTC when creating the loan, it uses the value specified by the loan request. maxLTC is just the max value it can't exceed. There's nothing forcing the operator to

accept every loan at/under the maxLTC. The operator would have discretion and wouldn't clear bad loans that are undercollateralized from the start. The only thing it's doing it providing a hard stop for the ltc.

```
(
    uint256 amount,
    uint256 interest,
    uint256 ltc,
    uint256 duration,
) = cooler.requests(id);


// Validate terms
if (interest < minimumInterest)
    revert InterestMinimum();
if (ltc > maxLTC)
    revert LTCMaximum();
```

We see in the above lines that it pulls the requested LTC from the cooler and compares it to maxLTC to confirm it isn't too high. It doesn't use the maxLTC value, it simply functions as a guardrail to make sure it doesn't accidentally clear a bad loan to gives way too much DAI. Technically if the price of gOHM was to increase dramatically the contract would become useless, but it wouldn't lead to any lost/stuck funds and it would be easy enough to just deploy a new contract with different bounds.

This should be low severity.

You've created a valid escalation for 11 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation rejected

After careful consideration & internal discussion. Sherlock did not have full context on the intent of the contract. Based on certain assumptions, & as pointed out by the escalation, in case of price change, the contract does not seem to serve its purpose and is rendered useless with the same bounds, making this issue a valid medium.

sherlock-admin

Escalation rejected

> After careful consideration & internal discussion. Sherlock did not have full context on the intent of the contract. Based on certain assumptions, & as pointed out by the escalation, in case of price change, the contract does not seem to serve its purpose and is rendered useless with the same bounds, making this issue a valid medium.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.