SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Teller
Prepared by: Sherlock
Lead Security Expert: 0x52
Dates Audited: April 14 - April 22, 2023
Prepared on: June 19, 2023

Introduction

Lend, borrow, and launch the FinTechs of tomorrow. Teller brings major financial marketplaces to DeFi.

Scope

Repository: teller-protocol/teller-protocol-v2

Branch: develop

Commit: cb66c9e348cdf1fd6d9b0416a49d663f5b6a693c

For the detailed scope, see the contest details.

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
15	5

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

immeas
cducrest-brainbot
0x52
cccz
RaymondFam
0xbepresent
branch_indigo
T1MOH
carrotsmuggler
J4de
jpserrat
dingo

innertia
saidam017
spyrosonic10
dipp
chaduke
Bauer
dacian
monrel
nobody2018
yixxas
HonorLt
0xmuxyz
MiloTruck
duc
whoismatthewmc1
BAHOZ
juancito
ctf_sec
shaka
evmboi32
mahdikarimi
xAlismx
ubl4nk
Ruhum
Nyx
Fanz
whiteh4t9527
sinarette
mrpathfindr
ck
jasonxiale
0xGoodess
deadrxsezzz
__141345__
giovannidisiena
yshhjain
8olidity
tallo
PawelK
n33k
Breeje
tsvetanovv
0x2e
ak1
Saeedalipoor01988
hake
Dug
HexHackers
tvdung94
Vagner
Delvir0
Inspex

Issue H-1: CollateralManager#commitCollateral can be called on an active loan

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/168

Found by

0x52, 0xbepresent, dipp, innertia, jpserrat, spyrosonic10

Summary

CollateralManager#commitCollateral never checks if the loan has been accepted allowing users to add collaterals after which can DOS the loan.

Vulnerability Detail

CollateralManager.sol#L117-L130

```
function commitCollateral(
    uint256 _bidId,
    Collateral[] calldata _collateralInfo
) public returns (bool validation_) {
    address borrower = tellerV2.getLoanBorrower(_bidId);
    (validation_, ) = checkBalances(borrower, _collateralInfo); <- @audit-issue
    ↪  never checks that loan isn't active

    if (validation_) {
        for (uint256 i; i < _collateralInfo.length; i++) {
            Collateral memory info = _collateralInfo[i];
            _commitCollateral(_bidId, info);
        }
    }
}
```

CollateralManager#commitCollateral does not contain any check that the bidId is pending or at least that it isn't accepted. This means that collateral can be committed to an already accepted bid, modifying bidCollaterals.

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L393-L409

```
function _withdraw(uint256 _bidId, address _receiver) internal virtual {
    for (
        uint256 i;
        i < _bidCollaterals[_bidId].collateralAddresses.length();
        i++
```

```
    ) {
        // Get collateral info
        Collateral storage collateralInfo = _bidCollaterals[_bidId]
            .collateralInfo[
                _bidCollaterals[_bidId].collateralAddresses.at(i)
            ];
        // Withdraw collateral from escrow and send it to bid lender
        ICollateralEscrowV1(_escrows[_bidId]).withdraw(
            collateralInfo._collateralAddress,
            collateralInfo._amount,
            _receiver
        );
```

bidCollaterals is used to trigger the withdrawal from the escrow to the receiver, which closing the loan and liquidations. This can be used to DOS a loan AFTER it has already been filled.

1) User A creates a bid for 10 ETH against 50,000 USDC at 10% APR
2) User B sees this bid and decides to fill it
3) After the loan is accepted, User A calls CollateralManager#commitCollateral with a malicious token they create
4) User A doesn't pay their loan and it becomes liquidatable
5) User B calls liquidate but it reverts when the escrow attempts to transfer out the malicious token
6) User A demands a ransom to return the funds
7) User A enables the malicious token transfer once the ransom is paid

Impact

Loans can be permanently DOS'd even after being accepted

Code Snippet

CollateralManager.sol#L117-L130

CollateralManager.sol#L138-L147

Tool used

Manual Review

Recommendation

CollateralManager#commitCollateral should revert if loan is active.

Discussion

ethereumdegen

This seems to be a duplicate of the issue that states that the commitCollateral is publicly callable. A fix is being implemented which will make that function only callable by TellerV2.sol contract which should remedy this vulnerability. Thank you .

dugdaniels

Escalate for 10 USDC

As the sponsor already stated, this should be marked as a duplicate of https://github.com/sherlock-audit/2023-03-teller-judging/issues/169

They are the same root cause and the same attack vector.

sherlock-admin

> Escalate for 10 USDC
>
> As the sponsor already stated, this should be marked as a duplicate of https://github.com/sherlock-audit/2023-03-teller-judging/issues/169
>
> They are the same root cause and the same attack vector.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

Trumpero

I believe this issue is different from #169, and they have different root causes. This issue demonstrates that the `commitCollateral` function can be called on the active loan/ after the accepted bid, to add a malicious token and block liquidation, even when the caller is legitimate (the borrower in scenario).

ethereumdegen

This is fixed by the same fix as 169 fix/sherlock/169

hrishibhat

Escalation rejected

This is a different issue with a different root cause from #169 as pointed out by the Lead Judge

sherlock-admin

> Escalation rejected

This is a different issue with a different root cause from #169 as pointed out by the Lead Judge

```
This issue's escalations have been rejected!

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

IAm0x52

Fixed here by restricting commitCollateral to only be called by TellerV2, which will never call commitCollateral on an active loan

Issue H-2: CollateralManager#commitCollateral can be called by anyone

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/169

Found by

0x52, 0xbepresent, 8olidity, BAHOZ, HonorLt, Inspex, J4de, MiloTruck, PawelK, Ruhum, __141345__, cccz, cducrest-brainbot, chaduke, ctf_sec, duc, evmboi32, giovannidisiena, immeas, jpserrat, juancito, mahdikarimi, nobody2018, shaka, sinarette, ubl4nk, whoismatthewmc1, xAlismx, yshhjain

Summary

CollateralManager#commitCollateral has no access control allowing users to freely add malicious tokens to any bid

Vulnerability Detail

CollateralManager.sol#L117-L130

```
function commitCollateral(
    uint256 _bidId,
    Collateral[] calldata _collateralInfo
) public returns (bool validation_) {
    address borrower = tellerV2.getLoanBorrower(_bidId);
    (validation_, ) = checkBalances(borrower, _collateralInfo); <- @audit-issue
    ↪  no access control

    if (validation_) {
        for (uint256 i; i < _collateralInfo.length; i++) {
            Collateral memory info = _collateralInfo[i];
            _commitCollateral(_bidId, info);
        }
    }
}
```

CollateralManager#commitCollateral has no access control and can be called by anyone on any bidID. This allows an attacker to front-run lenders and add malicious tokens to a loan right before it is filled.

1) A malicious user creates a malicious token that can be transferred once before being paused and returns uint256.max for balanceOf
2) User A creates a loan for 10e18 ETH against 50,000e6 USDC at 10% APR
3) User B decides to fill this loan and calls TellerV2#lenderAcceptBid

4) The malicious user sees this and front-runs with a CollateralManager#commitCollateral call adding the malicious token
5) Malicious token is now paused breaking both liquidations and fully paying off the loan
6) Malicious user leverages this to ransom the locked tokens, unpausing when it is paid

Impact

User can add malicious collateral calls to any bid they wish

Code Snippet

CollateralManager.sol#L117-L130

CollateralManager.sol#L138-L147

Tool used

Manual Review

Recommendation

Cause CollateralManager#commitCollateral to revert if called by anyone other than the borrower, their approved forwarder or TellerV2

Discussion

ethereumdegen

Thank you to the many many auditors who discovered this vulnerability. Will fix.

ethereumdegen

Github PR : fix/sherlock/169

IAm0x52

PR: #98

IAm0x52

Fix looks good. commitCollateral can now only be called by TellerV2

Issue H-3: CollateralManager#commitCollateral overwrites collateralInfo._amount if called with an existing collateral

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/170

Found by

0x52, BAHOZ, Ruhum, carrotsmuggler, cccz, cducrest-brainbot, chaduke, ctf_sec, dingo, evmboi32, jasonxiale, jpserrat, juancito, mahdikarimi, ubl4nk, whiteh4t9527, whoismatthewmc1, xAlismx

Summary

When duplicate collateral is committed, the collateral amount is overwritten with the new value. This allows borrowers to front-run bid acceptance to change their collateral and steal from lenders.

Vulnerability Detail

CollateralManager.sol#L426-L442

```
function _commitCollateral(
    uint256 _bidId,
    Collateral memory _collateralInfo
) internal virtual {
    CollateralInfo storage collateral = _bidCollaterals[_bidId];
    collateral.collateralAddresses.add(_collateralInfo._collateralAddress);
    collateral.collateralInfo[
        _collateralInfo._collateralAddress
    ] = _collateralInfo; <- @audit-issue collateral info overwritten
    emit CollateralCommitted(
        _bidId,
        _collateralInfo._collateralType,
        _collateralInfo._collateralAddress,
        _collateralInfo._amount,
        _collateralInfo._tokenId
    );
}
```

When a duplicate collateral is committed it overwrites the collateralInfo for that token, which is used to determine how much collateral to escrow from the borrower.

TellerV2.sol#L470-L484

```
function lenderAcceptBid(uint256 _bidId)
    external
```

```
    override
    pendingBid(_bidId, "lenderAcceptBid")
    whenNotPaused
    returns (
        uint256 amountToProtocol,
        uint256 amountToMarketplace,
        uint256 amountToBorrower
    )
{
    // Retrieve bid
    Bid storage bid = bids[_bidId];

    address sender = _msgSenderForMarket(bid.marketplaceId);
```

TellerV2#lenderAcceptBid only allows the lender input the bidId of the bid they wish to accept, not allowing them to specify the expected collateral. This allows lenders to be honeypot and front-run causing massive loss of funds:

1) Malicious user creates and commits a bid to take a loan of 10e18 ETH against 100,000e6 USDC with 15% APR
2) Lender sees this and calls TellerV2#lenderAcceptBid
3) Malicious user front-runs transaction with commitCollateral call setting USDC to 1
4) Bid is filled sending malicious user 10e18 ETH and escrowing 1 USDC
5) Attacker doesn't repay loan and has stolen 10e18 ETH for the price of 1 USDC

Impact

Bid acceptance can be front-run to cause massive losses to lenders

Code Snippet

TellerV2.sol#L470-L558

Tool used

Manual Review

Recommendation

Allow lender to specify collateral info and check that it matches the committed addresses and amounts

Discussion

Trumpero

The lenderAcceptBid function will revert if it tries to deposit duplicated collateral because the CollateralEscrowV1.depositAsset function has a requirement to avoid asset overwriting (https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscrowV1.sol#L66-L70). Therefore, if the borrower front-runs and changes the collateral amount, the lenderAcceptBid function will revert, causing the lender to lose gas only.

ethereumdegen

as trumpero says, this issue has been known and so we put in reversions to prevent this from happening. However we do want to eventually fix this in a better way so that you will be able to use two of the same ERC721 as collateral for one loan at some point. Ex. using two bored apes as collateral for one loan. This is not possible due to our patch for this.

0xJuancito

Escalate for 10 USDC

The validness and the `High` severity of this issue is proved with a **coded POC** in https://github.com/sherlock-audit/2023-03-teller-judging/issues/250.

> The lenderAcceptBid function will revert if it tries to deposit duplicated collateral because the CollateralEscrowV1.depositAsset function has a requirement to avoid asset overwriting (https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscrowV1.sol#L66-L70). Therefore, if the borrower front-runs and changes the collateral amount, the lenderAcceptBid function will revert, causing the lender to lose gas only.

The comment addresses the `deposit` action, but the attack is actually performed before that, during `commitCollateral`. Commited collateral can actually be updated, which is the root cause of this issue.

1. A malicious borrower submits a bid (collateral is commited but not deposited).
2. A victim lender decides to accept the bid and sends the tx to the mempool.
3. The malicious borrower sees the tx in the mempool and frontruns it by updating the commited collateral to a minimum.
4. The accept bid tx from the victim lender succeeds and the borrower collateral is finally deposited (but just a minimum).

Take note that funds are deposited **once** via deployAndDeposit, which is called during the lenderAcceptBid function.

The impact is High because assets are lent with a minimum collateral provided (of as low as 1 wei). This can be consider stealing users funds.

A coded proof is provided in https://github.com/sherlock-audit/2023-03-teller-judging/issues/250 , which I also suggest to be used for the final report, as it demonstrates the issue with a POC and provides coded recommendations to mitigate the issue.

sherlock-admin

> Escalate for 10 USDC
>
> The validness and the `High` severity of this issue is proved with a **coded POC** in https://github.com/sherlock-audit/2023-03-teller-judging/issues/250.
>
> > The lenderAcceptBid function will revert if it tries to deposit duplicated collateral because the CollateralEscrowV1.depositAsset function has a requirement to avoid asset overwriting (https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscrowV1.sol#L66-L70). Therefore, if the borrower front-runs and changes the collateral amount, the lenderAcceptBid function will revert, causing the lender to lose gas only.
>
> The comment addresses the `deposit` action, but the attack is actually performed before that, during `commitCollateral`. Commited collateral can actually be updated, which is the root cause of this issue.
>
> 1. A malicious borrower submits a bid (collateral is commited but not deposited).
> 2. A victim lender decides to accept the bid and sends the tx to the mempool.
> 3. The malicious borrower sees the tx in the mempool and frontruns it by updating the commited collateral to a minimum.
> 4. The accept bid tx from the victim lender succeeds and the borrower collateral is finally deposited (but just a minimum).
>
> Take note that funds are deposited **once** via deployAndDeposit, which is called during the lenderAcceptBid function.
>
> The impact is High because assets are lent with a minimum collateral provided (of as low as 1 wei). This can be consider stealing users funds.
>
> A coded proof is provided in https://github.com/sherlock-audit/2023-03-teller-judging/issues/250 , which I also suggest to be used for the final

report, as it demonstrates the issue with a POC and provides coded recommendations to mitigate the issue.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

cducrest

The `CollateralManager` will never attempt to deposit duplicate collateral, the code for committing collateral is the following:

```
function _commitCollateral(
    uint256 _bidId,
    Collateral memory _collateralInfo
) internal virtual {
    CollateralInfo storage collateral = _bidCollaterals[_bidId];
    collateral.collateralAddresses.add(_collateralInfo._collateralAddress);
    collateral.collateralInfo[
        _collateralInfo._collateralAddress
    ] = _collateralInfo;
    ...
}
```

The set `collateral.collateralAddresses` will not contain the address of the collateral twice since OZ's `EnumerableSetUpgradeable` does not add an element to a set if it is already present (it's a set, not a list). The previous value for `collateral.collateralInfo[_collateralInfo._collateralAddress]` will be replaced with no additional side-effect.

Hence the check for duplicate collateral in `CollateralEscrowV1` is never triggered:

```
function depositAsset(
    CollateralType _collateralType,
    address _collateralAddress,
    uint256 _amount,
    uint256 _tokenId
) external payable virtual onlyOwner {
    require(_amount > 0, "Deposit amount cannot be zero");
    _depositCollateral(
        _collateralType,
        _collateralAddress,
        _amount,
        _tokenId
    );
    Collateral storage collateral = collateralBalances[_collateralAddress];
```

```
    //Avoids asset overwriting.  Can get rid of this restriction by
↪   restructuring collateral balances storage so it isnt a mapping based on
↪   address.
    require(
        collateral._amount == 0,  // @audit this check is never triggered
        "Unable to deposit multiple collateral asset instances of the same
↪   contract address."
    );

    collateral._collateralType = _collateralType;
    collateral._amount = _amount;
    collateral._tokenId = _tokenId;
    emit CollateralDeposited(_collateralAddress, _amount);
}
```

See OZ's docs

ethereumdegen

Your comment about '@audit this check is never triggered' is disproven by this test which had already been in the repo.

```
function test_depositAsset_ERC721_double_collateral_overwrite_prevention()
    public
{
    CollateralEscrowV1_Override escrow = CollateralEscrowV1_Override(
        address(borrower.getEscrow())
    );

    uint256 tokenIdA = erc721Mock.mint(address(borrower));
    uint256 tokenIdB = erc721Mock.mint(address(borrower));

    borrower.approveERC721(address(erc721Mock), tokenIdA);
    borrower.approveERC721(address(erc721Mock), tokenIdB);

    vm.prank(address(borrower));
    escrow.depositAsset(
        CollateralType.ERC721,
        address(erc721Mock),
        1,
        tokenIdA
    );

    uint256 storedBalance = borrower.getBalance(address(erc721Mock));

    assertEq(storedBalance, 1, "Escrow deposit unsuccessful");
```

```
    vm.expectRevert(
        "Unable to deposit multiple collateral asset instances of the same
        ↪  contract address."
    );
    vm.prank(address(borrower));
    escrow.depositAsset(
        CollateralType.ERC721,
        address(erc721Mock),
        1,
        tokenIdB
    );
}
```

hrishibhat

Escalation accepted

Valid high After further review and discussions, this is a valid high issue and the POC in #250 proves the same.

sherlock-admin

> Escalation accepted
>
> Valid high After further review and discussions, this is a valid high issue and the POC in #250 proves the same.

```
This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on
↪  this issue.
```

IAm0x52

PR: #101

IAm0x52

Fix looks good. Commit will revert if collateral is already present

Issue H-4: _repayLoan will fail if lender is blacklisted

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/212

Found by

T1MOH, carrotsmuggler, cccz, cducrest-brainbot

Summary

The internal function that repays a loan `_repayLoan` attempts to transfer the loan token back to the lender. If the loan token implements a blacklist like the common USDC token, the transfer may be impossible and the repayment will fail.

This internal `_repayLoan` function is called during any partial / full repayment and during liquidation.

Vulnerability Detail

The function to repay the loan to the lender directly transfers the token to the lender:

```
function _repayLoan(...) internal virtual {
    ...
    bid.loanDetails.lendingToken.safeTransferFrom(
        _msgSenderForMarket(bid.marketplaceId),
        lender,
        paymentAmount
    );
    ...
```

This function is called by any function that attemps to repay a loan (including liquidate): https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L593 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L615 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L649 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L690

Any of these functions will fail if loan lender is blacklisted by the token.

During repayment the loan lender is computed by:

```
function getLoanLender(uint256 _bidId)
    public
    view
```

```
    returns (address lender_)
{
    lender_ = bids[_bidId].lender;

    if (lender_ == address(lenderManager)) {
        return lenderManager.ownerOf(_bidId);
    }
}
```

If the lender controls a blacklisted address, they can use the lenderManager to selectively transfer the loan to / from the blacklisted whenever they want.

Impact

Any lender can prevent repayment of a loan and its liquidation. In particular, a lender can wait until a loan is almost completely repaid, transfer the loan to a blacklisted address (even one they do not control) to prevent the loan to be fully repaid / liquidated. The loan will default and borrower will not be able to withdraw their collateral.

This result in a guaranteed griefing attack on the collateral of a user.

If the lender controls a blacklisted address, they can additionally withdraw the collateral of the user.

I believe the impact is high since the griefing attack is always possible whenever lent token uses a blacklist, and results in a guaranteed loss of collateral.

Code Snippet

The function to withdraw collateral only works when loan is paid or transfer to lender when loan is defaulted:

```
function withdraw(uint256 _bidId) external {
    BidState bidState = tellerV2.getBidState(_bidId);
    if (bidState == BidState.PAID) {
        _withdraw(_bidId, tellerV2.getLoanBorrower(_bidId));
    } else if (tellerV2.isLoanDefaulted(_bidId)) {
        _withdraw(_bidId, tellerV2.getLoanLender(_bidId));
        emit CollateralClaimed(_bidId);
    } else {
        revert("collateral cannot be withdrawn");
    }
}
```

Tool used

Manual Review

Recommendation

Use a push/pull pattern for transferring tokens. Allow repayment of loan and withdraw the tokens of the user into `TellerV2` (or an escrow) and allow lender to withdraw the repayment from `TellerV2` (or the escrow). This way, the repayment will fail only if `TellerV2` is blacklisted.

Discussion

ethereumdegen

We feel as though this is technically a valid issue but

1) hopefully it is extremely rare and edge-case
2) it is so rare that it doesnt justify such a large change to the protocol and reducing UX flow for the entire system (an extra tx to withdraw funds)
3) and finally, borrowers just have to be aware of the extra risk posed when using tokens which have 'denylists' in them or freezing capabilities and also should justify the risks of the particular party acting as the lender (if lender is likely to have assets frozen, do not borrow from them, etc.) .

IAm0x52

Sponsor has acknowledged this risk

ethereumdegen

Okay so after further review we did make a PR to make an option to pay into an escrow vault. This escrow vault is a new contract, a simple smart contract wallet to hold funds for a lender in the event that their account is not able to receive tokens. This way, borrowers can repay loans regardless.
PR: https://github.com/teller-protocol/teller-protocol-v2/pull/105

IAm0x52

Fix looks good. Contract now utilizes an escrow contract that will receive tokens in the event that the lender is unable to receive the repayment directly. This ensures that repayment is always possible.

jacksanford1

Teller decided to fix this issue, so changing label to "Will Fix".

Issue H-5: Malicious user can abuse UpdateCommitment to create commitments for other users

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/260

Found by

0x52, 0xbepresent, Bauer, J4de, dingo, immeas

Summary

UpdateCommitment checks that the original lender is msg.sender but never validates that the original lender == new lender. This allows malicious users to effectively create a commitment for another user, allowing them to drain funds from them.

Vulnerability Detail

LenderCommitmentForwarder.sol#L208-L224

```
function updateCommitment(
    uint256 _commitmentId,
    Commitment calldata _commitment
) public commitmentLender(_commitmentId) { <- @audit-info checks that lender is
↪  msg.sender
    require(
        _commitment.principalTokenAddress ==
            commitments[_commitmentId].principalTokenAddress,
        "Principal token address cannot be updated."
    );
    require(
        _commitment.marketId == commitments[_commitmentId].marketId,
        "Market Id cannot be updated."
    );

    commitments[_commitmentId] = _commitment; <- @audit-issue never checks
    ↪  _commitment.lender

    validateCommitment(commitments[_commitmentId]);
```

UpdateCommitment is intended to allow users to update their commitment but due to lack of verification of _commitment.lender, a malicious user create a commitment then update it to a new lender. By using bad loan parameters they can steal funds from the attacker user.

Impact

UpdateCommitment can be used to create a malicious commitment for another user and steal their funds

Code Snippet

LenderCommitmentForwarder.sol#L208-L233

Tool used

Manual Review

Recommendation

Check that the update lender is the same the original lender

Discussion

ethereumdegen

Thank you for this feedback. This is a high severity issue as it could be used to unexpectedly steal tokens that another use had previously approved to the contract. Will fix.

passabilities

https://github.com/teller-protocol/teller-protocol-v2/pull/67

IAm0x52

Fix looks good. _commitment.lender (updated lender) is now required to be msg.sender

Issue M-1: lender could be forced to withdraw collateral even if he/she would rather wait for liquidation during default

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/2

Found by

0xGoodess, 0xbepresent, MiloTruck, Nyx, cducrest-brainbot, chaduke, ctf_sec, duc, innertia

Summary

lender could be forced to withdraw collateral even if he/she would rather wait for liquidation during default

Vulnerability Detail

CollateralManager.withdraw would pass if the loan is defaulted (the borrower does not pay interest in time); in that case, anyone can trigger an withdrawal on behalf of the lender before the liquidation delay period passes.

withdraw logic from CollateralManager.

```
 * @notice Withdraws deposited collateral from the created escrow of a bid that
 ↪  has been successfully repaid.
 * @param _bidId The id of the bid to withdraw collateral for.
 */
function withdraw(uint256 _bidId) external {
    BidState bidState = tellerV2.getBidState(_bidId);
    console2.log("WITHDRAW %d", uint256(bidState));
    if (bidState == BidState.PAID) {
        _withdraw(_bidId, tellerV2.getLoanBorrower(_bidId));
    } else if (tellerV2.isLoanDefaulted(_bidId)) { @> audit
        _withdraw(_bidId, tellerV2.getLoanLender(_bidId));
        emit CollateralClaimed(_bidId);
    } else {
        revert("collateral cannot be withdrawn");
    }
}
```

Impact

anyone can force lender to take up collateral during liquidation delay and liquidation could be something that never happen. This does not match the intention based on

the spec which implies that lender has an option: `3) When the loan is fully repaid, the borrower can withdraw the collateral. If the loan becomes defaulted instead, then the lender has a 24 hour grace period to claim the collateral (losing the principal)`

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L250-L260

Tool used

Manual Review

Recommendation

check that the caller is the lender

```
    function withdraw(uint256 _bidId) external {
        BidState bidState = tellerV2.getBidState(_bidId);
        console2.log("WITHDRAW %d", uint256(bidState));
        if (bidState == BidState.PAID) {
            _withdraw(_bidId, tellerV2.getLoanBorrower(_bidId));
        } else if (tellerV2.isLoanDefaulted(_bidId)) {
+++         uint256 _marketplaceId = bidState.marketplaceId;
+++         address sender = _msgSenderForMarket(_marketplaceId);
+++         address lender = tellerV2.getLoanLender(_bidId);
+++         require(sender == lender, "sender must be the lender");
            _withdraw(_bidId, lender);
            emit CollateralClaimed(_bidId);
        } else {
            revert("collateral cannot be withdrawn");
        }
    }
```

Discussion

ethereumdegen

Thank you for the feedback i will review this with the team.

ctf-sec

Escalate for 10 USDC. Impact is high, so the severity is not medium but high.

> anyone can force lender to take up collateral during liquidation delay and liquidation could be something that never happen.

because the lack of access control dispute normal liquidation flow, which is clearly a high impact

https://docs.sherlock.xyz/audits/judging/judging

> High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

Permissionlessly dispute the liquidation flow make "the cost of the attack low" and generate bad debts, which equal to "loss of fund". Clearly this is "not considered an acceptable risk by a reasonable protocol team"

Thanks

sherlock-admin

> Escalate for 10 USDC. Impact is high, so the severity is not medium but high.
>
> > anyone can force lender to take up collateral during liquidation delay and liquidation could be something that never happen.
>
> because the lack of access control dispute normal liquidation flow, which is clearly a high impact
>
> https://docs.sherlock.xyz/audits/judging/judging
>
> > High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> Permissionlessly dispute the liquidation flow make "the cost of the attack low" and generate bad debts, which equal to "loss of fund". Clearly this is "not considered an acceptable risk by a reasonable protocol team"
>
> Thanks

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

Trumpero

I believe it is a medium since the lender will receive the collateral, which holds more value than the loan itself usually. If the collateral holds less value than the loan, then no one, except the lender, will be willing to liquidate this loan. The lender only miss out on the opportunity to be repaid the loan as originally expected.

ethereumdegen

Github PR: Issue 224 - Access control on collateral withdraw

hrishibhat

Escalation rejected

Valid medium As the Lead judge pointed out The impact of this issue is a medium. There is no loss of funds for the protocol, this is about being able to withdraw collateral in case the loan is defaulted during the liquidation delay. A missed out opportunity to be repaid, not a high issue

sherlock-admin

> Escalation rejected
>
> Valid medium As the Lead judge pointed out The impact of this issue is a medium. There is no loss of funds for the protocol, this is about being able to withdraw collateral in case the loan is defaulted during the liquidation delay. A missed out opportunity to be repaid, not a high issue

```
This issue's escalations have been rejected!

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

IAm0x52

Fix looks good. Collateral can now only be withdrawn by the appropriate party. Borrower if paid in full or lender/owner of loan NFT if defaulted.

Issue M-2: The calculation time methods of `calculateNextDueDate` and `_canLiquidateLoan` are inconsistent

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/78

Found by

J4de, immeas

Summary

The calculation time methods of `calculateNextDueDate` and `_canLiquidateLoan` are inconsistent

Vulnerability Detail

```
File: TellerV2.sol
 854     function calculateNextDueDate(uint256 _bidId)
 855         public
 856         view
 857         returns (uint32 dueDate_)
 858     {
 859         Bid storage bid = bids[_bidId];
 860         if (bids[_bidId].state != BidState.ACCEPTED) return dueDate_;
 861
 862         uint32 lastRepaidTimestamp = lastRepaidTimestamp(_bidId);
 863
 864         // Calculate due date if payment cycle is set to monthly
 865         if (bidPaymentCycleType[_bidId] == PaymentCycleType.Monthly) {
 866             // Calculate the cycle number the last repayment was made
 867             uint256 lastPaymentCycle = BPBDTL.diffMonths(
 868                 bid.loanDetails.acceptedTimestamp,
 869
```

The `calculateNextDueDate` function is used by the borrower to query the date of the next repayment. Generally speaking, the borrower will think that as long as the repayment is completed at this point in time, the collateral will not be liquidated.

```
File: TellerV2.sol
 953     function _canLiquidateLoan(uint256 _bidId, uint32 _liquidationDelay)
 954         internal
 955         view
 956         returns (bool)
 957     {
 958         Bid storage bid = bids[_bidId];
```

```
959
960         // Make sure loan cannot be liquidated if it is not active
961         if (bid.state != BidState.ACCEPTED) return false;
962
963         if (bidDefaultDuration[_bidId] == 0) return false;
964
965         return (uint32(block.timestamp) -
966             _liquidationDelay -
967             lastRepaidTimestamp(_bidId) >
968             bidDefaultDuration[_bidId]);
969     }
```

However, when the `_canLiquidateLoan` function actually judges whether it can be liquidated, the time calculation mechanism is completely different from that of `calculateNextDueDate` function, which may cause that if the time point calculated by `_canLiquidateLoan` is earlier than the time point of `calculateNextDueDate` function, the borrower may also be liquidated in the case of legal repayment.

Borrowers cannot query the specific liquidation time point, but can only query whether they can be liquidated through the `isLoanDefaulted` function or `isLoanLiquidateable` function. When they query that they can be liquidated, they may have already been liquidated.

Impact

Borrowers may be liquidated if repayments are made on time.

Code Snippet

https://github.com/teller-protocol/teller-protocol-v2/blob/cb66c9e348cdf1fd6d9b0416a49d663f5b6a693c/packages/contracts/contracts/TellerV2.sol#L953-L969

Tool used

Manual Review

Recommendation

It is recommended to verify that the liquidation time point cannot be shorter than the repayment period and allow users to query the exact liquidation time point.

Discussion

ethereumdegen

Github PR: Issue 494 - improving logic for is loan liquidateable

IAm0x52

Fix looks good. Liquidation logic has been revised to align it with dueDate

Issue M-3: updateCommitmentBorrowers does not delete all existing users

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/88

Found by

cducrest-brainbot, monrel, nobody2018

Summary

`delete` a complex structure that includes mapping will cause problem. See [ethereum/solidity#11843](https://github.com/ethereum/solidity/pull/11843) for more info.

Vulnerability Detail

The lender can update the list of borrowers by calling `LenderCommitmentForwarder.updateCommitmentBorrowers`. The list of borrowers is EnumerableSetUpgradeable.AddressSet that is a complex structure containing mapping. Using the `delete` keyword to delete this structure will not erase the mapping inside it. Let's look at the code of this function.

```
mapping(uint256 => EnumerableSetUpgradeable.AddressSet)
        internal commitmentBorrowersList;

function updateCommitmentBorrowers(
        uint256 _commitmentId,
        address[] calldata _borrowerAddressList
    ) public commitmentLender(_commitmentId) {
        delete commitmentBorrowersList[_commitmentId];
        _addBorrowersToCommitmentAllowlist(_commitmentId, _borrowerAddressList);
    }
```

I wrote a similar function to prove the problem.

```
using EnumerableSet for EnumerableSet.AddressSet;
    mapping(uint256 => EnumerableSet.AddressSet) internal users;

    function test_deleteEnumerableSet() public {
        uint256 id = 1;
        address[] memory newUsers = new address[](2);
        newUsers[0] = address(0x1);
        newUsers[1] = address(0x2);

        for (uint256 i = 0; i < newUsers.length; i++) {
            users[id].add(newUsers[i]);
```

```
        }
        delete users[id];
        newUsers[0] = address(0x3);
        newUsers[1] = address(0x4);
        for (uint256 i = 0; i < newUsers.length; i++) {
            users[id].add(newUsers[i]);
        }
        bool exist = users[id].contains(address(0x1));
        if(exist) {
            emit log_string("address(0x1) exist");
        }
        exist = users[id].contains(address(0x2));
        if(exist) {
            emit log_string("address(0x2) exist");
        }
    }
/*
[PASS] test_deleteEnumerableSet() (gas: 174783)
Logs:
  address(0x1) exist
  address(0x2) exist
*/
```

Impact

The deleted Users can still successfully call `LenderCommitmentForwarder.acceptCommitment` to get a loan.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/LenderCommitmentForwarder.sol#L240-L246

Tool used

Manual Review

Recommendation

In order to clean an `EnumerableSet`, you can either remove all elements one by one or create a fresh instance using an array of `EnumerableSet`.

Discussion

ethereumdegen

Github PR :
Issue 88 - Changing the way borrowers are updated for Lender Commitment

IAm0x52

Fix looks good. updateCommitmentBorrowers has been split into two separate functions:

addCommitmentBorrowers - Explicitly adds borrowers listed
removeCommitmentBorrowers - Explicitly removes borrowers listed

Issue M-4: If the collateral is a fee-on-transfer token, repayment will be blocked

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/91

Found by

0x2e, 8olidity, BAHOZ, Bauer, Breeje, Delvir0, HexHackers, HonorLt, MiloTruck, Nyx, Vagner, __141345__, ak1, cccz, cducrest-brainbot, ck, ctf_sec, dacian, deadrxsezzz, dingo, duc, evmboi32, giovannidisiena, innertia, monrel, n33k, nobody2018, saidam017, shaka, sinarette, spyrosonic10, tsvetanovv, tvdung94, whiteh4t9527, yixxas

Summary

As we all know, some tokens will deduct fees when transferring token. In this way, **the actual amount of token received by the receiver will be less than the amount sent**. If the collateral is this type of token, the amount of collateral recorded in the contract will bigger than the actual amount. **When the borrower repays the loan, the amount of collateral withdrawn will be insufficient, causing tx revert**.

Vulnerability Detail

The `_bidCollaterals` mapping of `CollateralManager` records the `CollateralInfo` of each bidId. This structure records the collateral information provided by the user when creating a bid for a loan. A lender can accept a loan by calling `TellerV2.lenderAcceptBid` that will eventually transfer the user's collateral from the user address to the CollateralEscrowV1 contract corresponding to the loan. The whole process will deduct fee twice.

```
//CollateralManager.sol
function _deposit(uint256 _bidId, Collateral memory collateralInfo)
        internal
        virtual
    {
        ......
        // Pull collateral from borrower & deposit into escrow
        if (collateralInfo._collateralType == CollateralType.ERC20) {
            IERC20Upgradeable(collateralInfo._collateralAddress).transferFrom(
↪   //transferFrom first time
                borrower,
                address(this),
                collateralInfo._amount
            );
            IERC20Upgradeable(collateralInfo._collateralAddress).approve(
```

```
                escrowAddress,
                collateralInfo._amount
            );
            collateralEscrow.depositAsset(      //transferFrom second time
                CollateralType.ERC20,
                collateralInfo._collateralAddress,
                collateralInfo._amount,         //this value is from user's input
                0
            );
        }
        ......
    }
```

The amount of collateral recorded by the CollateralEscrowV1 contract is equal to the amount originally submitted by the user.

When the borrower repays the loan, `collateralManager.withdraw` will be triggered. This function internally calls `CollateralEscrowV1.withdraw`. Since the balance of the collateral in the CollateralEscrowV1 contract is less than the amount to be withdrawn, the entire transaction reverts.

```
//CollateralEscrowV1.sol
function _withdrawCollateral(
        Collateral memory _collateral,
        address _collateralAddress,
        uint256 _amount,
        address _recipient
    ) internal {
        // Withdraw ERC20
        if (_collateral._collateralType == CollateralType.ERC20) {
            IERC20Upgradeable(_collateralAddress).transfer(   //revert
                _recipient,
                _collateral._amount //_collateral.balanceOf(address(this)) <
↪  _collateral._amount
            );
        }
    ......
    }
```

Impact

The borrower's collateral is stuck in the instance of CollateralEscrowV1. Non-professional users will never know that they need to manually transfer some collateral into CollateralEscrowV1 to successfully repay.

- This issue blocked the user's repayment, causing the loan to be liquidated.

- The liquidator will not succeed by calling `TellerV2.liquidateLoanFull`.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L323-L326

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L432-L434

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L510

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L327-L341

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscrowV1.sol#L73

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscrowV1.sol#L166-L169

Tool used

Manual Review

Recommendation

Two ways to fix this issue.

- The `afterBalance-beforeBalance` method should be used when recording the amount of collateral.
- ```
  --- a/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscr
  ↪  owV1.sol
  +++ b/teller-protocol-v2/packages/contracts/contracts/escrow/CollateralEscr
  ↪  owV1.sol
  @@ -165,7 +165,7 @@ contract CollateralEscrowV1 is OwnableUpgradeable,
  ↪  ICollateralEscrowV1 {
           if (_collateral._collateralType == CollateralType.ERC20) {
               IERC20Upgradeable(_collateralAddress).transfer(
                   _recipient,
  -                _collateral._amount
  +
  ↪  IERC20Upgradeable(_collateralAddress).balanceOf(address(this))
               );
           }
  ```
  ```

Discussion

ethereumdegen

This is the same as the issue that I explained in the readme for this contest about 'poisoned collateral'. It has been known previously that collateral in the escrow could be made non-transferrable which makes loan repayment impossible. Thank you for the report. Thisis a re-iteration of what was stated as 'known issues' in the contest readme.

iamjakethehuman

Escalate for 10 USDC Disagree with sponsor's comment. This is not the case described in the Readme. The Readme states:

> A: Known issue 1: Collateral assets that can be 'paused' for transfer do exhibit a risk since they may not be able to be withdrawn from the loans as collateral. Furthermore, collateral assets that can be made non-transferrable can actually 'poison' a collateral vault and make a loan non-liquidateable since a liquidateLoan call would revert.

The contest FAQ states:

> FEE-ON-TRANSFER: any

The project clearly hasn't implemented logic for fee-on-transfer tokens when it has clearly stated that it should be able to operate with such tokens. Would love to hear judge's opinion on this.

sherlock-admin

> Escalate for 10 USDC Disagree with sponsor's comment. This is not the case described in the Readme. The Readme states:
>
> > A: Known issue 1: Collateral assets that can be 'paused' for transfer do exhibit a risk since they may not be able to be withdrawn from the loans as collateral. Furthermore, collateral assets that can be made non-transferrable can actually 'poison' a collateral vault and make a loan non-liquidateable since a liquidateLoan call would revert.
>
> The contest FAQ states:
>
> > FEE-ON-TRANSFER: any
>
> The project clearly hasn't implemented logic for fee-on-transfer tokens when it has clearly stated that it should be able to operate with such tokens. Would love to hear judge's opinion on this.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

securitygrid

According to README.md: Collateral should not be able to be permanently locked/burned in the contracts. So this issue is valid M.

Love4codes

yeah, i second it's a valid M. Let's see the judges opinion tho

ethereumdegen

Github PR: Issue 225 - Separate logic for repay and collateral withdraw

hrishibhat

Escalation accepted

Valid medium This issue is a valid medium as fee-on-transfer tokens are in scope.

sherlock-admin

> Escalation accepted
>
> Valid medium This issue is a valid medium as fee-on-transfer tokens are in scope.

```
This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on
↪  this issue.
```

IAm0x52

Fix looks good. Repaying a loan in full no longer forces withdrawing collateral, preventing the repay call from reverting

Issue M-5: Lender can take borrower's collateral before first payment due

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/92

Found by

cducrest-brainbot, dacian

Summary

For PaymentCycleType.Seconds if PaymentDefault < PaymentCycle, Lender can take Borrower's collateral before first payment is due. If PaymentDefault > 0 but very small, Lender can do this almost immediately after accepting borrower's bid. This is especially bad as the Market Operator who controls these parameters can also be the Lender.

Vulnerability Detail

Lender calls CollateralManager.withdraw() L254, which calls TellerV2.isLoanDefaulted() L930, which bypasses the 1 day grace period & doesn't take into account when first payment is due.

Impact

Borrower loses their collateral before they can even make their first repayment, almost instantly if PaymentDefault > 0 but very small.

Code Snippet

Put this test in TellerV2_Test.sol:

```
function test_LenderQuicklyTakesCollateral() public {
    MarketRegistry mReg =
↪   (MarketRegistry)(payable(address(tellerV2.marketRegistry())));

    // payment cycle 3600 seconds, payment default 1 second
    // payment will be in default almost immediately upon being
    // accepted, even though the first payment is not due for much longer
    // than the default time
    uint32 PAYMENT_CYCLE_SEC    = 3600;
    uint32 PAYMENT_DEFAULT_SEC = 1;

    vm.startPrank(address(marketOwner));
    mReg.setPaymentCycle(marketId1, PaymentCycleType.Seconds, PAYMENT_CYCLE_SEC);
```

```
    mReg.setPaymentDefaultDuration(marketId1, PAYMENT_DEFAULT_SEC);
    vm.stopPrank();

    //Submit bid as borrower
    uint256 bidId = submitCollateralBid();
    // Accept bid as lender
    acceptBid(bidId);

    // almost immediately take the collateral as the lender, even though
    // the first payment wasn't due for much later
    ICollateralManager cMgr = tellerV2.collateralManager();
    skip(PAYMENT_DEFAULT_SEC+1);
    cMgr.withdraw(bidId);
    // try again to verify the collateral has been taken
    vm.expectRevert("No collateral balance for asset");
    cMgr.withdraw(bidId);
}
```

Tool used

Manual Review

Recommendation

Change the calculations done as a consequence of calling TellerV2.isLoanDefaulted() to take into account when first payment is due; see similar code which does this TellerV2.calculateNextDueDate() L886-L899. Lender should only be able to take Borrower's collateral after the Borrower has missed their first payment deadline by PaymentDefault seconds.

Consider enforcing sensible minimums for PaymentDefault. If PaymentDefault = 0 no liquidations will ever be possible as TellerV2._canLiquidateLoan() L963 will always return false, so perhaps it shouldn't be possible to set PaymentDefault = 0.

Discussion

devdacian

Escalate for 10 USDC

In previous audit contests, the issue of the Borrower being prematurely liquidated/having their collateral seized has been judged as a High finding:

1) High - Loan can be written off by anybody before overdue delay expires - Union Finance, Sherlock
2) High - Users can be liquidated prematurely because calculation understates value of underlying position Blueberry, Sherlock

3) High - Lender is able to seize the collateral by changing the loan parameters Abra, C4
4) High - Users may be liquidated right after taking maximal debt Papr, C4

Please explain the objective standard of truth that when applied to these previous submissions where Borrowers are prematurely liquidated/have their collateral seized judges them all as High, but when applied to my submission where Borrowers are prematurely liquidated/have their collateral seized judges it only medium.

Alternatively, please judge my submission as a High finding in line with the historical judging in both Sherlock & C4.

sherlock-admin

> Escalate for 10 USDC
>
> In previous audit contests, the issue of the Borrower being prematurely liquidated/having their collateral seized has been judged as a High finding:
>
> 1) High - Loan can be written off by anybody before overdue delay expires - Union Finance, Sherlock
> 2) High - Users can be liquidated prematurely because calculation understates value of underlying position Blueberry, Sherlock
> 3) High - Lender is able to seize the collateral by changing the loan parameters Abra, C4
> 4) High - Users may be liquidated right after taking maximal debt Papr, C4
>
> Please explain the objective standard of truth that when applied to these previous submissions where Borrowers are prematurely liquidated/have their collateral seized judges them all as High, but when applied to my submission where Borrowers are prematurely liquidated/have their collateral seized judges it only medium.
>
> Alternatively, please judge my submission as a High finding in line with the historical judging in both Sherlock & C4.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

Trumpero

This scenario only occurs when the PaymentDefaultDuration variable has been set too low by the admin/owner (1 in the POC), so I believe it is an unlikely occurrence, and should be a medium issue. It is not similar to the other cases mentioned by Watson in the past contests, as they do not require any manipulations/mistakes from the admin.

hrishibhat

Escalation rejected

Valid medium This requires an admin error/malice to set a low default duration to cause loss of funds. As opposed to the other past issues mentioned in the escalation have a logical flaw in the code to cause loss of funds.

sherlock-admin

> Escalation rejected
>
> Valid medium This requires an admin error/malice to set a low default duration to cause loss of funds. As opposed to the other past issues mentioned in the escalation have a logical flaw in the code to cause loss of funds.

```
This issue's escalations have been rejected!

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

ethereumdegen

This is a non-issue because borrowers will be able to validate the default duration of a loan before agreeing to it. If they agree to a very short default duration, that is their choice. It does make sense to enforce a default duration longer than 1 payment cycle for ux purposes but i see that as an optional 'nice to have.' it will just be important to clearly communicate the default duration to borrowers.

IAm0x52

Sponsor has acknowledged this risk

Issue M-6: LenderCommitmentForwarder#updateCommitment can be front-run by malicious borrower to cause lender to over-commit funds

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/176

Found by

0x52, chaduke

Summary

This is the same idea as approve vs increaseAlllowance. updateCommitment is a bit worse though because there are more reason why a user may wish to update their commitment (expiration, collateral ratio, interest rate, etc).

Vulnerability Detail

LenderCommitmentForwarder.sol#L212-L222

```
require(
    _commitment.principalTokenAddress ==
        commitments[_commitmentId].principalTokenAddress,
    "Principal token address cannot be updated."
);
require(
    _commitment.marketId == commitments[_commitmentId].marketId,
    "Market Id cannot be updated."
);

commitments[_commitmentId] = _commitment;
```

LenderCommitmentForwarder#updateCommitment overwrites ALL of the commitment data. This means that even if a user is calling it to update even one value the maxPrincipal will reset, opening up the following attack vector:

1) User A creates a commitment for 100e6 USDC lending against ETH
2) User A's commitment is close to expiry so they call to update their commitment with a new expiration
3) User B sees this update and front-runs it with a loan against the commitment for 100e6 USDC
4) User A's commitment is updated and the amount is set back to 100e6 USDC
5) User B takes out another loan for 100e6 USDC

6) User A has now loaned out 200e6 USDC when they only meant to loan 100e6 USDC

Impact

Commitment is abused to over-commit lender

Code Snippet

LenderCommitmentForwarder.sol#L208-L233

Tool used

Manual Review

Recommendation

Create a function that allows users to extend expiry while keeping amount unchanged. Additionally create a function similar to increaseApproval which increase amount instead of overwriting amount.

Discussion

ethereumdegen

I think the only correct way to solve this is to never decrement or change the 'max principal' amount on a commitment because no matter how that is done it can be frontrun attacked in this way.

The only viable solution is to add a mapping that keeps track of how much capital has been allocated from a commitment and always make sure that that is LEQ than the maxPrincipal for that commitment.

ethereumdegen

Github PR: Issue 176 - fixing frontrun attackability of commitment maxPrincipal

IAm0x52

Fix looks good. Now tracks the amount of allocated principle instead of decrementing the max principal of the commitment.

Issue M-7: Bid submission vulnerable to market parameters changes

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/205

Found by

BAHOZ, Fanz, T1MOH, cducrest-brainbot, ck, immeas, jpserrat, juancito, whoismatthewmc1

Summary

The details for the audit state:

> Market owners should NOT be able to race-condition attack borrowers or lenders by changing market settings while bids are being submitted or accepted (while tx are in mempool). Care has been taken to ensure that this is not possible (similar in theory to sandwich attacking but worse as if possible it could cause unexpected and non-consentual interest rate on a loan) and further-auditing of this is welcome.

However, there is little protection in place to protect the submitter of a bid from changes in market parameters.

Vulnerability Detail

In _submitBid(), certain bid parameters are taken from the `marketRegistry`:

```
function _submitBid(...)
    ...
    (bid.terms.paymentCycle, bidPaymentCycleType[bidId]) = marketRegistry
        .getPaymentCycle(_marketplaceId);

    bid.terms.APR = _APR;

    bidDefaultDuration[bidId] = marketRegistry.getPaymentDefaultDuration(
        _marketplaceId
    );

    bidExpirationTime[bidId] = marketRegistry.getBidExpirationTime(
        _marketplaceId
    );

    bid.paymentType = marketRegistry.getPaymentType(_marketplaceId);

    bid.terms.paymentCycleAmount = V2Calculations
```

```
            .calculatePaymentCycleAmount(
                bid.paymentType,
                bidPaymentCycleType[bidId],
                _principal,
                _duration,
                bid.terms.paymentCycle,
                _APR
            );
    ...
```

All the parameters taken from marketRegistry are controlled by the market owner: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/MarketRegistry.sol#L487-L511

Impact

If market parameters are changed in between the borrower submitting a bid transaction and the transaction being applied, borrower may be subject to changes in `bidDefaultDuration`, `bidExpirationTime`, `paymentType`, `paymentCycle`, `bidPaymentCycleType` and `paymentCycleAmount`.

That is, the user may be committed to the bid for longer / shorter than expected. They may have a longer / shorter default duration (time for the loan being considered defaulted / eligible for liquidation). They have un-provisioned for payment type and cycle parameters.

I believe most of this will have a medium impact on borrower (mild inconveniences / resolvable by directly repaying the loan) if the market owner is not evil and adapting the parameters reasonably.

An evil market owner can set the value of `bidDefaultDuration` and `paymentCycle` very low (0) so that the loan will default immediately. It can then accept the bid, make user default immediately, and liquidate the loan to steal the user's collateral. This results in a loss of collateral for the borrower.

Code Snippet

Tool used

Manual Review

Recommendation

Take every single parameters as input of `_submitBid()` (including fee percents) and compare them to the values in `marketRegistry` to make sure borrower agrees with them, revert if they differ.

Discussion

ethereumdegen

The way the protocol was designed, market owners are 'trusted' however we will add a new submitBid method that adds these checks in case we want to support trustless market owners in the future

ethereumdegen

Github PR: Issue 205 - Harden submitBid from frontrun attacks by market owner

ethereumdegen

A fix was made but it is not being applied at this time because it would create too drastic of a change to the ABI, the way that contracts interact with submitBid. Furthermore, at the time, market owners are 'trusted' and so there is not a concern that they will frontrun.

When we do decide to open up the ability for anyone to be a market owner, we will revisit this fix and implement a protection strategy against front running the fees in the way or add a hard-cap to the fee such as 10% in order to not impact the ABI.

IAm0x52

Sponsor has acknowledged this risk

Issue M-8: EMI last payment not handled perfectly could lead to borrower losing collaterals

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/315

Found by

RaymondFam

Summary

The ternary logic of `calculateAmountOwed()` could have the last EMI payment under calculated, leading to borrower not paying the owed principal and possibly losing the collaterals if care has not been given to.

Vulnerability Detail

Supposing Bob has a loan duration of 100 days such that the payment cycle is evenly spread out, i.e payment due every 10 days, here is a typical scenario:

1. Bob has been making his payment due on time to avoid getting marked delinquent. For the last payment due, Bob decides to make it 5 minutes earlier just to make sure he will not miss it.
2. However, `duePrincipal_` ends up assigned the minimum of `owedAmount - interest_` and `owedPrincipal_`, where the former is chosen since `oweTime` is less than `_bid.terms.paymentCycle`:

```
} else {
    // Default to PaymentType.EMI
    // Max payable amount in a cycle
    // NOTE: the last cycle could have less than the calculated payment amount
    uint256 maxCycleOwed = isLastPaymentCycle
        ? owedPrincipal_ + interest_
        : _bid.terms.paymentCycleAmount;

    // Calculate accrued amount due since last repayment
    uint256 owedAmount = (maxCycleOwed * owedTime) /
        _bid.terms.paymentCycle;
    duePrincipal_ = Math.min(owedAmount - interest_, owedPrincipal_);
}
```

3. Hence, in `_repayLoan()`, `paymentAmount >= _owedAmount` equals false failing to close the loan to have the collaterals returned to Bob:

```
if (paymentAmount >= _owedAmount) {
    paymentAmount = _owedAmount;
    bid.state = BidState.PAID;

    // Remove borrower's active bid
    _borrowerBidsActive[bid.borrower].remove(_bidId);

    // If loan is is being liquidated and backed by collateral, withdraw and
↪   send to borrower
    if (_shouldWithdrawCollateral) {
        collateralManager.withdraw(_bidId);
    }

    emit LoanRepaid(_bidId);
```

4. While lingering and not paying too much attention to the collateral still in escrow, Bob presumes his loan is now settled.
5. Next, Alex the lender has been waiting for this golden opportunity and proceeds to calling CollateralManager.withdraw() to claim all collaterals as soon as the loan turns defaulted.

Impact

Bob ended up losing all collaterals for the sake of the minute amount of loan unpaid whereas Alex receives almost all principal plus interests on top of the collaterals.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.sol#L107-L119

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L727-L739

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L250-L260

Tool used

Manual Review

Recommendation

Consider refactoring the affected ternary logic as follows:

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.sol#L107-L119

```
        } else {
+            duePrincipal = isLastPaymentCycle
+                ? owedPrincipal
+                : (_bid.terms.paymentCycleAmount * owedTime) /
 ↪  _bid.terms.paymentCycle;

            // Default to PaymentType.EMI
            // Max payable amount in a cycle
            // NOTE: the last cycle could have less than the calculated payment
 ↪  amount
-            uint256 maxCycleOwed = isLastPaymentCycle
-                ? owedPrincipal_ + interest_
-                : _bid.terms.paymentCycleAmount;

            // Calculate accrued amount due since last repayment
-            uint256 owedAmount = (maxCycleOwed * owedTime) /
-                _bid.terms.paymentCycle;
-            duePrincipal_ = Math.min(owedAmount - interest_, owedPrincipal_);
        }
```

Discussion

iamjakethehuman

Escalate for 10 USDC Disagree with severity. Issue should be considered QA/ Low. Upon repaying, the user will see whether the final repayment has been successful. In fact, a corresponding event is emitted. If the final repayment was successful `LoanRepaid(_bidId)` is emitted. If the collateral isn't withdrawn `LoanRepayment(_bidId)` is emitted. Also, the said scenario requires the user to not see that they haven't received their collateral back for at least `bidDefaultDuration[_bidId])` which makes the scenario even more unrealistic. https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L965-#L968

sherlock-admin

> Escalate for 10 USDC Disagree with severity. Issue should be considered QA/ Low. Upon repaying, the user will see whether the final repayment has been successful. In fact, a corresponding event is emitted. If the final repayment was successful `LoanRepaid(_bidId)` is emitted. If the collateral isn't withdrawn `LoanRepayment(_bidId)` is emitted. Also, the said scenario requires the user to not see that they haven't received their collateral back for at least `bidDefaultDuration[_bidId])` which makes the scenario even more unrealistic.

> https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L965-#L968

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

raymondfam

Most borrowers would form a habit making the recurring payments in time by doing it slightly earlier just like anyone would be making a car, credit card or house payment just to make sure no late payments could tarnish their credit reputation. The protocol logic is intuitive in making adjustments to users forming this good habit. However, the current ternary logic could trap/trick many unsavvy borrowers when handling the last payment. The adept users might notice the event emitted but would run into the same issue again unless function repayLoanFull is used instead of function repayLoan. As a borrower, I would expect the system handle the transaction all clean come the last payment.

Additionally, function isLoanDefaulted does not care how much principal and interest has been paid in the past; all it cares is whether or not `(uint32(block.timestamp) - _liquidationDelay - lastRepaidTimestamp(_bidId) > bidDefaultDuration[_bidId])` where `_liquidationDelay == 0` in this case. Depending on the value of `bidDefaultDuration[_bidId]`, the amount of seconds elapsed could transpire quickly. Imagine doing this before sleep, and then getting caught up with a full day of chores and work the next day, and before too long the position is already deemed delinquent without the borrower even aware of it.

ethereumdegen

Your updated logic does compile, however it makes 5 tests fail. Are you also implying that these 5 tests are incorrect? Or does that mean that there is an issue with your suggested logic? In any case, more deeply investigating.

[FAIL. Reason: Assertion failed.] test_01_calculateAmountOwed() (gas: 442923)
[FAIL. Reason: Assertion failed.] test_02_calculateAmountOwed() (gas: 483615)
[FAIL. Reason: Assertion failed.] test_03_calculateAmountOwed() (gas: 336240)
[FAIL. Reason: Assertion failed.] test_04_calculateAmountOwed() (gas: 441542)
[FAIL. Reason: Assertion failed.] test_05_calculateAmountOwed() (gas: 335700)

hrishibhat

Escalation rejected

Valid medium This is a valid issue where it gives an incorrect amount during the last payment cycle. Sponsor comment:

> is most certainly a valid issue, there is a fix . The issue is if a user makes

> all their payments normally and on time, then in the last payment cycle, their amount owed fluctutes over time however the intent is that in the last payment cycle the amount owed should be 'the entire remaining balance' . Not fluctuating over time. (the time fluctuation is because of code that, for cycles that arent the last payment cycle, amount owed is (current cycle amount + amount unpaid from previous cycles))

sherlock-admin

> Escalation rejected
>
> Valid medium This is a valid issue where it gives an incorrect amount during the last payment cycle. Sponsor comment:
>
> > is most certainly a valid issue, there is a fix . The issue is if a user makes all their payments normally and on time, then in the last payment cycle, their amount owed fluctutes over time however the intent is that in the last payment cycle the amount owed should be 'the entire remaining balance' . Not fluctuating over time. (the time fluctuation is because of code that, for cycles that arent the last payment cycle, amount owed is (current cycle amount + amount unpaid from previous cycles))

```
This issue's escalations have been rejected!

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

ethereumdegen

PR fix: https://github.com/teller-protocol/teller-protocol-v2/pull/99

IAm0x52

Fix looks good. owedAmount calculation has been slightly modified so that it will won't final payment amount by owedTime which will result in a complete repayment of the principal

Issue M-9: defaulting doesn't change the state of the loan

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/317

Found by

Dug, MiloTruck, RaymondFam, cccz, cducrest-brainbot, deadrxsezzz, immeas, jpserrat, mrpathfindr, sinarette

Summary

The lender can claim the borrowers collateral in case they have defaulted on their payments. This however does not change the state of the loan so the borrower can continue making payments to the lender even though the loan is defaulted.

Vulnerability Detail

If a loan defaults the lender (or anyone) can seize the collateral and send it to the lender: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L254-L257

```
File: CollateralManager.sol

254:         } else if (tellerV2.isLoanDefaulted(_bidId)) {
255:             _withdraw(_bidId, tellerV2.getLoanLender(_bidId)); // sends
 ↪  collateral to lender
256:             emit CollateralClaimed(_bidId);
257:         } else {
```

Since this is in `CollateralManager` nothing is updating the state kept in `TellerV2` which will still be `ACCEPTED`. The lender could still make payments (in vain).

Impact

The borrower can continue paying unknowing that the loan is defaulted. The lender could, given a defaulted loan, see that the lender is trying to save their loan and front run the late payment with a seize of collateral. Then get both the late payment and the collateral. This is quite an unlikely scenario though.

The loan will also be left active since even if the borrower pays the `withdraw` of collateral will fail since the collateral is no longer there.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L254-L257

Tool used

Manual Review

Recommendation

Remove the possibility for the lender to default the loan in `CollateralManager`.
Move defaulting to `TellerV2` so it can properly close the loan.

Discussion

ethereumdegen

PR (draft): https://github.com/teller-protocol/teller-protocol-v2/pull/103
(also see issue 311)

IAm0x52

Confirming this fix for 311/317. Looks good. When a lender claims collateral from an overdue loan, the status of the loan is changed to "CLOSED" preventing any further repayment from the borrower.

jacksanford1

Changed label to Will Fix since Teller ended up making a fix (and 0x52 reviewed it).

Issue M-10: bids can be created against markets that doesn't exist

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/323

Found by

immeas, saidam017

Summary

Bids can be created against markets that does not yet exist. When this market is created, the bid can be accepted but neither defaulted/liquidated nor repaid.

Vulnerability Detail

There's no verification that the market actually exists when submitting a bid. Hence a user could submit a bid for a non existing market.

For it to not revert it must have 0% APY and the bid cannot be accepted until a market exists.

However, when this market is created the bid can be accepted. Then the loan would be impossible to default/liquidate:

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L963

```
File: TellerV2.sol

963:         if (bidDefaultDuration[_bidId] == 0) return false;
```

Since `bidDefaultDuration[_bidId]` will be `0`

Any attempt to repay will revert due to division by 0:

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.sol#L116-L117

```
File: libraries/V2Calculations.sol

116:             uint256 owedAmount = (maxCycleOwed * owedTime) /
117:                 _bid.terms.paymentCycle;
```

Since `_bid.terms.paymentCycle` will also be `0` (and it will always end up in this branch since `PaymentType` will be `EMI (0)`).

Hence the loan can never be closed.

PoC:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { UpgradeableBeacon } from
↪  "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";

import { TellerV2 } from "../contracts/TellerV2.sol";
import { CollateralManager } from "../contracts/CollateralManager.sol";
import { LenderCommitmentForwarder } from
↪  "../contracts/LenderCommitmentForwarder.sol";
import { CollateralEscrowV1 } from "../contracts/escrow/CollateralEscrowV1.sol";
import { MarketRegistry } from "../contracts/MarketRegistry.sol";

import { ReputationManagerMock } from
↪  "../contracts/mock/ReputationManagerMock.sol";
import { LenderManagerMock } from "../contracts/mock/LenderManagerMock.sol";
import { TellerASMock } from "../contracts/mock/TellerASMock.sol";

import {TestERC20Token} from "./tokens/TestERC20Token.sol";

import "lib/forge-std/src/Test.sol";
import "lib/forge-std/src/StdAssertions.sol";

contract LoansTest is Test {
    MarketRegistry marketRegistry;
    TellerV2 tellerV2;

    TestERC20Token principalToken;

    address alice = address(0x1111);
    address bob = address(0x2222);
    address owner = address(0x3333);

    function setUp() public {
        tellerV2 = new TellerV2(address(0));

        marketRegistry = new MarketRegistry();
        TellerASMock tellerAs = new TellerASMock();
        marketRegistry.initialize(tellerAs);

        LenderCommitmentForwarder lenderCommitmentForwarder =
            new
↪  LenderCommitmentForwarder(address(tellerV2),address(marketRegistry));
        CollateralManager collateralManager = new CollateralManager();
        collateralManager.initialize(address(new UpgradeableBeacon(address(new
↪  CollateralEscrowV1()))),
```

```
            address(tellerV2));
        address rm = address(new ReputationManagerMock());
        address lm = address(new LenderManagerMock());
        tellerV2.initialize(0, address(marketRegistry), rm,
↪   address(lenderCommitmentForwarder),
            address(collateralManager), lm);

        principalToken = new TestERC20Token("Principal Token", "PRIN", 12e18,
↪   18);
    }

    function testSubmitBidForNonExistingMarket() public {
        uint256 amount = 12e18;
        principalToken.transfer(bob,amount);

        vm.prank(bob);
        principalToken.approve(address(tellerV2),amount);

        // alice places bid on non-existing market
        vm.prank(alice);
        uint256 bidId = tellerV2.submitBid(
            address(principalToken),
            1, // non-existing right now
            amount,
            360 days,
            0, // any APY != 0 will cause revert on div by 0
            "",
            alice
        );

        // bid cannot be accepted before market
        vm.expectRevert(); // div by 0
        vm.prank(bob);
        tellerV2.lenderAcceptBid(bidId);

        vm.startPrank(owner);
        uint256 marketId = marketRegistry.createMarket(
            owner,
            30 days,
            30 days,
            1 days,
            0,
            false,
            false,
            ""
        );
        marketRegistry.setMarketFeeRecipient(marketId, owner);
        vm.stopPrank();
```

```
        // lender takes bid
        vm.prank(bob);
        tellerV2.lenderAcceptBid(bidId);

        // should be liquidatable now
        vm.warp(32 days);

        // loan cannot be defaulted/liquidated
        assertFalse(tellerV2.isLoanDefaulted(bidId));
        assertFalse(tellerV2.isLoanLiquidateable(bidId));

        vm.startPrank(alice);
        principalToken.approve(address(tellerV2),12e18);

        // and loan cannot be repaid
        vm.expectRevert(); // division by 0
        tellerV2.repayLoanFull(bidId);
        vm.stopPrank();
    }
}
```

Impact

This will lock any collateral forever since there's no way to retrieve it. For this to happen accidentally a borrower would have to create a bid for a non existing market with 0% APY though.

This could also be used to lure lenders since the loan cannot be liquidated/defaulted. This might be difficult since the APY must be 0% for the bid to be created. Also, this will lock any collateral provided by the borrower forever.

Due to these circumstances I'm categorizing this as medium.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L334-L411

Tool used

Manual Review

Recommendation

When submitting a bid, verify that the market exists.

Discussion

ethereumdegen

A user creating a bid for a market that does not yet exist yet COULD exist in the future is potentially a concern. For example an attacker could see that there are bids open for market 88, create markets until market 88 exists , and then fulfill those loans with whatever rules they want. Our user interface on the front end will prevent bids from being created with an invalid market ID so in reality this should not be an issue but in solidity strictly yes this is a valid issue. Thank you.

ethereumdegen

We should make a function name isMarketOpen that verifies that 1) the marketId is less than the number of markets and 2) the market is not closed and we should use that in submitBid instead of !isMarketClosed

iamjakethehuman

Escalate for 10 USDC The issue requires the borrower to intentionally send his assets to a non-existing market. This would be the same as mistakenly sending the assets to a wrong address. Issues requiring the user to use wrong input have historically been invalidated/ considered QA. Furthermore, it also requires the said market to have 0% APY which is unrealistic, as there is no incentive for such markets to exist.

sherlock-admin

> Escalate for 10 USDC The issue requires the borrower to intentionally send his assets to a non-existing market. This would be the same as mistakenly sending the assets to a wrong address. Issues requiring the user to use wrong input have historically been invalidated/ considered QA. Furthermore, it also requires the said market to have 0% APY which is unrealistic, as there is no incentive for such markets to exist.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

said017

Escalate for 10 USDC

This is valid issue.

This scenario not only applied with borrower accidentally create to a non-existing market.

Consider this scenario, malicious borrower listen to `marketRegistry's createMarket()` market creation. The `marketId` can easily be seen in mempool or

can be predicted since it only increment of the previous `marketId`.

The attacker then front run the market creation and create malicious borrow offer trough TellerV2 's `submitBid()` and provide the soon to be created `marketId`.

0% APY is part of malicious borrower input so it is realistic.

This is valid issue, caused by the relatively easy and likely setup, and also caused the following impact :

1. Malicious borrower can create borrow offer in trusted and exclusive marketplace before it is created, make clueless lender that thought the marketplace is legit take the bid.
2. And the more harmful impact is the borrow offer can't be liquidated nor defaulted since `bidDefaultDuration[_bidId]` will be 0 and checking if bid can be liquidated or defaulted always return false.

sherlock-admin

> Escalate for 10 USDC
>
> This is valid issue.
>
> This scenario not only applied with borrower accidentally create to a non-existing market.
>
> Consider this scenario, malicious borrower listen to `marketRegistry's createMarket()` market creation. The `marketId` can easily be seen in mempool or can be predicted since it only increment of the previous `marketId`.
>
> The attacker then front run the market creation and create malicious borrow offer trough TellerV2 's `submitBid()` and provide the soon to be created `marketId`.
>
> 0% APY is part of malicious borrower input so it is realistic.
>
> This is valid issue, caused by the relatively easy and likely setup, and also caused the following impact :
>
> 1. Malicious borrower can create borrow offer in trusted and exclusive marketplace before it is created, make clueless lender that thought the marketplace is legit take the bid.
> 2. And the more harmful impact is the borrow offer can't be liquidated nor defaulted since `bidDefaultDuration[_bidId]` will be 0 and checking if bid can be liquidated or defaulted always return false.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

ethereumdegen

Github PR: Issue 323 - add check for is market open

hrishibhat

Escalation accepted

Valid medium Accepting the second escalaiton Based on the above comments, This is a valid issue confirmed by the sponsor and the attack path in the 2nd escalation is possible and confirmed by the Sponsor too.

sherlock-admin

> Escalation accepted
>
> Valid medium Accepting the second escalaiton Based on the above comments, This is a valid issue confirmed by the sponsor and the attack path in the 2nd escalation is possible and confirmed by the Sponsor too.

```
This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on
↪  this issue.
```

IAm0x52

Fix looks good. Creates and utilizes a new check called "isMarketOpen" which requires that specified market exists

Issue M-11: last repayments are calculated incorrectly for "irregular" loan durations

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/328

Found by

immeas

Summary

When taking a loan, a borrower expects that at the end of each payment cycle they should pay `paymentCycleAmount`. This is not true for loans that are not a multiple of `paymentCycle`.

Vulnerability Detail

Imagine a loan of `1000` that is taken for 2.5 payment cycles (skip interest to keep calculations simple).

A borrower would expect to pay `400 + 400 + 200`

This holds true for the first installment.

But lets look at what happens at the second installment, here's the calculation of what is to pay in `V2Calculations.sol`:

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.sol#L93-L101

```
File: libraries/V2Calculations.sol

 93:         // Cast to int265 to avoid underflow errors (negative means loan
 ↪  duration has passed)
 94:         int256 durationLeftOnLoan = int256(
 95:             uint256(_bid.loanDetails.loanDuration)
 96:         ) -
 97:             (int256(_timestamp) -
 98:                 int256(uint256(_bid.loanDetails.acceptedTimestamp)));
 99:         bool isLastPaymentCycle = durationLeftOnLoan <
100:             int256(uint256(_bid.terms.paymentCycle)) || // Check if current
 ↪  payment cycle is within or beyond the last one
101:             owedPrincipal_ + interest_ <= _bid.terms.paymentCycleAmount; //
 ↪  Check if what is left to pay is less than the payment cycle amount
```

Simplified the first calculation says `timeleft = loanDuration - (now - acceptedTimestamp)` and then if `timeleft < paymentCycle` we are within the last

payment cycle.

This isn't true for loan durations that aren't multiples of the payment cycles. This code says the last payment cycle is when you are one payment cycle from the end of the loan. Which is not the same as last payment cycle as my example above shows.

PoC:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { UpgradeableBeacon } from
↪  "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";

import { AddressUpgradeable } from
↪  "@openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol";

import { TellerV2 } from "../contracts/TellerV2.sol";
import { Payment } from "../contracts/TellerV2Storage.sol";
import { CollateralManager } from "../contracts/CollateralManager.sol";
import { LenderCommitmentForwarder } from
↪  "../contracts/LenderCommitmentForwarder.sol";
import { CollateralEscrowV1 } from "../contracts/escrow/CollateralEscrowV1.sol";
import { Collateral, CollateralType } from
↪  "../contracts/interfaces/escrow/ICollateralEscrowV1.sol";

import { ReputationManagerMock } from
↪  "../contracts/mock/ReputationManagerMock.sol";
import { LenderManagerMock } from "../contracts/mock/LenderManagerMock.sol";
import { MarketRegistryMock } from "../contracts/mock/MarketRegistryMock.sol";

import {TestERC20Token} from "./tokens/TestERC20Token.sol";

import "lib/forge-std/src/Test.sol";

contract LoansTest is Test {
    using AddressUpgradeable for address;

    MarketRegistryMock marketRegistry;

    TellerV2 tellerV2;
    LenderCommitmentForwarder lenderCommitmentForwarder;
    CollateralManager collateralManager;

    TestERC20Token principalToken;

    address alice = address(0x1111);
```

```
    uint256 marketId = 0;

    function setUp() public {
        tellerV2 = new TellerV2(address(0));

        marketRegistry = new MarketRegistryMock();

        lenderCommitmentForwarder = new
↪   LenderCommitmentForwarder(address(tellerV2),address(marketRegistry));

        collateralManager = new CollateralManager();
        collateralManager.initialize(address(new UpgradeableBeacon(address(new
↪   CollateralEscrowV1()))), address(tellerV2));

        address rm = address(new ReputationManagerMock());
        address lm = address(new LenderManagerMock());
        tellerV2.initialize(0, address(marketRegistry), rm,
↪   address(lenderCommitmentForwarder), address(collateralManager), lm);

        marketRegistry.setMarketOwner(address(this));
        marketRegistry.setMarketFeeRecipient(address(this));

        tellerV2.setTrustedMarketForwarder(marketId,address(lenderCommitmentForw
↪   arder));

        principalToken = new TestERC20Token("Principal Token", "PRIN", 12e18,
↪   18);
    }


    function testLoanInstallmentsCalculatedIncorrectly() public {
        // payment cycle is 1000 in market registry

        uint256 amount = 1000;
        principalToken.transfer(alice,amount);

        vm.startPrank(alice);
        principalToken.approve(address(tellerV2),2*amount);
        uint256 bidId = tellerV2.submitBid(
            address(principalToken),
            marketId,
            amount,
            2500, // 2.5 payment cycles
            0, // 0 interest to make calculations easier
            "",
            alice
        );
        tellerV2.lenderAcceptBid(bidId);
```

```
        vm.stopPrank();

        // jump to first payment cycle end
        vm.warp(block.timestamp + 1000);
        Payment memory p = tellerV2.calculateAmountDue(bidId);
        assertEq(400,p.principal);

        // borrower pays on time
        vm.prank(alice);
        tellerV2.repayLoanMinimum(bidId);

        // jump to second payment cycle
        vm.warp(block.timestamp + 1000);
        p = tellerV2.calculateAmountDue(bidId);

        // should be 400 but is full loan
        assertEq(600,p.principal);
    }
}
```

The details of this finding are out of scope but since it makes `TellerV2`, in scope, behave unexpectedly I believe this finding to be in scope.

Impact

A borrower taking a loan might not be able to pay the last payment cycle and be liquidated. At the worst possible time since they've paid the whole loan on schedule up to the last installment. The liquidator just need to pay the last installment to take the whole collateral.

This requires the loan to not be a multiple of the payment cycle which might sound odd. But since a year is 365 days and a common payment cycle is 30 days I imagine there can be quite a lot of loans that after 360 days will end up in this issue.

There is also nothing stopping an unknowing borrower from placing a bid or accepting a commitment with an odd duration.

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.sol#L94-L101

Tool used

Manual Review

Recommendation

First I thought that you could remove the `lastPaymentCycle` calculation all together. I tried that and then also tested what happened with "irregular" loans with interest.

Then I found this in the EMI calculation:

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/libraries/NumbersLib.sol#L123

```
File: libraries/NumbersLib.sol

132:         uint256 n = Math.ceilDiv(loanDuration, cycleDuration);
```

EMI, which is designed for mortgages, assumes the payments is a discrete number of the same amortization essentially. I.e they don't allow "partial" periods at the end, because that doesn't make sense for a mortgage.

In Teller this is allowed which causes some issues with the EMI calculation since the above row will always round up to a full number of payment periods. If you also count interest, which triggers the EMI calculation: The lender, in an "irregular" loan duration, would get less per installment up to the last one which would be bigger. The funds would all be paid with the correct interest in the end just not in the expected amounts.

My recommendation now is:

either

- **don't allow loan durations that aren't a multiple of the period**, at least warn about it UI-wise, no one will lose any money but the installments might be split in unexpected amounts.
- **Do away with EMI all together** as DeFi loans aren't the same as mortgages. The defaulting/liquidation logic only cares about time since last payment.
- **Do more math** to make EMI work with irregular loan durations. This nerd sniped me:

More math:

From the link in the comment, https://en.wikipedia.org/wiki/Equated_monthly_installment you can follow one of the links in that wiki page to a derivation of the formula: http://rmathew.com/2006/calculating-emis.html

In the middle we have an equation which describes the owed amount at a time P_n:

$$P_n = Pt^n - E\frac{(t^n - 1)}{t - n}$$

where $t = 1 + r$ and r is the monthly interest rate ($apy * C/year$).

Now, from here, we want to calculate the loan at a time $P_{n+\Delta}$:

$$P_{n+\Delta} = Pt^n t_\Delta - E\frac{t^n - 1}{t - 1}t_\Delta - kE$$

Where k is c/C i.e. the ratio of partial cycle compared to a full cycle.

Same with t_Δ which is $1 + r_\Delta$, (r_Δ is also equal to kr, ratio of partial cycle rate to full cycle rate, which we'll use later).

Reorganize to get E from above:

$$E = Pr\frac{t^n t_\Delta}{t_\Delta \frac{t^n - 1}{t-1} + k}$$

Now substitute in $1 + r$ in place of t and $1 + r_\Delta$ instead of t_Δ and multiply both numerator and denominator with r:

$$E = P\frac{r(1 + r)^n(1 + r_\Delta)}{(1 + r_\Delta)((1 + r)^n - 1) + kr}$$

and $kr = r_\Delta$ gives us:

$$E = Pr(1 + r)^n\frac{(1 + r_\Delta)}{(1 + r_\Delta)((1 + r)^n - 1) + r_\Delta}$$

To check that this is correct, $r_\Delta = 0$ (no extra cycle added) should give us the regular EMI equation. Which we can see is true for the above. And $r_\Delta = r$ (a full extra cycle added) should give us the EMI equation but with $n + 1$ which we can also see it does.

Here are the code changes to use this, together with changes to `V2Calculations.sol` to calculate the last period correctly:

```
diff --git a/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calcula
↪  tions.sol
↪  b/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.s
↪  ol
index 1cce8da..1ad5bcf 100644
--- a/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.s
↪  ol
+++ b/teller-protocol-v2/packages/contracts/contracts/libraries/V2Calculations.s
↪  ol
@@ -90,30 +90,15 @@ library V2Calculations {
         uint256 owedTime = _timestamp - uint256(_lastRepaidTimestamp);
```

```
         interest_ = (interestOwedInAYear * owedTime) / daysInYear;

-        // Cast to int265 to avoid underflow errors (negative means loan
 ↪  duration has passed)
-        int256 durationLeftOnLoan = int256(
-            uint256(_bid.loanDetails.loanDuration)
-        ) -
-            (int256(_timestamp) -
-                int256(uint256(_bid.loanDetails.acceptedTimestamp)));
-        bool isLastPaymentCycle = durationLeftOnLoan <
-            int256(uint256(_bid.terms.paymentCycle)) || // Check if current
 ↪  payment cycle is within or beyond the last one
-            owedPrincipal_ + interest_ <= _bid.terms.paymentCycleAmount; //
 ↪  Check if what is left to pay is less than the payment cycle amount
-
         if (_bid.paymentType == PaymentType.Bullet) {
-            if (isLastPaymentCycle) {
-                duePrincipal_ = owedPrincipal_;
-            }
+            duePrincipal_ = owedPrincipal_;
         } else {
             // Default to PaymentType.EMI
             // Max payable amount in a cycle
             // NOTE: the last cycle could have less than the calculated payment
 ↪  amount
-            uint256 maxCycleOwed = isLastPaymentCycle
-                ? owedPrincipal_ + interest_
-                : _bid.terms.paymentCycleAmount;

             // Calculate accrued amount due since last repayment
-            uint256 owedAmount = (maxCycleOwed * owedTime) /
+            uint256 owedAmount = (_bid.terms.paymentCycleAmount * owedTime) /
                 _bid.terms.paymentCycle;
             duePrincipal_ = Math.min(owedAmount - interest_, owedPrincipal_);
         }
```

And then `NumbersLib.sol`:

```
diff --git
 ↪  a/teller-protocol-v2/packages/contracts/contracts/libraries/NumbersLib.sol
 ↪  b/teller-protocol-v2/packages/contracts/contracts/libraries/NumbersLib.sol
index f34dd9c..8ca48bc 100644
--- a/teller-protocol-v2/packages/contracts/contracts/libraries/NumbersLib.sol
+++ b/teller-protocol-v2/packages/contracts/contracts/libraries/NumbersLib.sol
@@ -120,7 +120,8 @@ library NumbersLib {
                 );
```

```
         // Number of payment cycles for the duration of the loan
-        uint256 n = Math.ceilDiv(loanDuration, cycleDuration);
+        uint256 n = loanDuration/ cycleDuration;
+        uint256 rest = loanDuration%cycleDuration;

         uint256 one = WadRayMath.wad();
         uint256 r = WadRayMath.pctToWad(apr).wadMul(cycleDuration).wadDiv(
@@ -128,8 +129,16 @@ library NumbersLib {
         );
         uint256 exp = (one + r).wadPow(n);
         uint256 numerator = principal.wadMul(r).wadMul(exp);
-        uint256 denominator = exp - one;

-        return numerator.wadDiv(denominator);
+        if(rest==0) {
+            // duration is multiple of cycle
+            uint256 denominator = exp - one;
+            return numerator.wadDiv(denominator);
+        }
+        // duration is an uneven cycle
+        uint256 rDelta =
 ↪  WadRayMath.pctToWad(apr).wadMul(rest).wadDiv(daysInYear);
+        uint256 n1 = numerator.wadMul(one + rDelta);
+        uint256 denom = ((one + rDelta).wadMul(exp - one)) + rDelta;
+        return n1.wadDiv(denom);
     }
 }
```

Discussion

ethereumdegen

It seems that in that example of a loan with 3 cycles, 400 then 400 and then 200, if the borrower is more than halfway through the second installment (second 400) , they would be considered to be in the 'lastPaymentCycle' incorrectly and would 'owe' 600 instead of 400. We will investigate this more for a fix.

cducrest

Escalate for 10 USDC

Issue is from out of scope contract. Both vulnerability and fix are in out-of-scope contract. Awarding issues for contracts not in scope incentivize poor scoping from protocol in the future and lead to more work for auditors with less pay (protocols pay for smaller scope and receive audits for bigger scopes).

Multiple incorrect behaviours could be spotted in `MarketRegistry` or `ReputationManager` as well but were not reported because they are out of scope.

sherlock-admin

> Escalate for 10 USDC
>
> Issue is from out of scope contract. Both vulnerability and fix are in out-of-scope contract. Awarding issues for contracts not in scope incentivize poor scoping from protocol in the future and lead to more work for auditors with less pay (protocols pay for smaller scope and receive audits for bigger scopes).
>
> Multiple incorrect behaviours could be spotted in `MarketRegistry` or `ReputationManager` as well but were not reported because they are out of scope.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

ethereumdegen

Github PR: Issue 328 - new repayment logic for v2 calculations

hrishibhat

Escalation rejected

Valid medium. The library is used in the in-scope contract and the error impacts an in-scope contract. This is a valid issue.

sherlock-admin

> Escalation rejected
>
> Valid medium. The library is used in the in-scope contract and the error impacts an in-scope contract. This is a valid issue.

```
This issue's escalations have been rejected!

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

IAm0x52

Fix looks good. Final repayment period is now calculated via modulo which allows it to correctly detect the final payment cycle for loans of irregular duration

Issue M-12: setLenderManager may cause some Lenders to lose their assets

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/339

Found by

MiloTruck, T1MOH, cccz, dingo, duc, saidam017, shaka, yixxas

Summary

If the contract's lenderManager changes, repaid assets will be sent to the old lenderManager

Vulnerability Detail

setLenderManager is used to change the lenderManager address of the contract

```
function setLenderManager(address _lenderManager)
    external
    reinitializer(8)
    onlyOwner
{
    _setLenderManager(_lenderManager);
}

function _setLenderManager(address _lenderManager)
    internal
    onlyInitializing
{
    require(
        _lenderManager.isContract(),
        "LenderManager must be a contract"
    );
    lenderManager = ILenderManager(_lenderManager);
}
```

claimLoanNFT will change the bid.lender to the current lenderManager

```
function claimLoanNFT(uint256 _bidId)
    external
    acceptedLoan(_bidId, "claimLoanNFT")
    whenNotPaused
{
    // Retrieve bid
    Bid storage bid = bids[_bidId];
```

```
    address sender = _msgSenderForMarket(bid.marketplaceId);
    require(sender == bid.lender, "only lender can claim NFT");
    // mint an NFT with the lender manager
    lenderManager.registerLoan(_bidId, sender);
    // set lender address to the lender manager so we know to check the owner of
↪   the NFT for the true lender
    bid.lender = address(lenderManager);
}
```

In getLoanLender, if the bid.lender is the current lenderManager, the owner of the NFT will be returned as the lender, and the repaid assets will be sent to the lender.

```
    function getLoanLender(uint256 _bidId)
        public
        view
        returns (address lender_)
    {
        lender_ = bids[_bidId].lender;

        if (lender_ == address(lenderManager)) {
            return lenderManager.ownerOf(_bidId);
        }
    }
...
        address lender = getLoanLender(_bidId);

        // Send payment to the lender
        bid.loanDetails.lendingToken.safeTransferFrom(
            _msgSenderForMarket(bid.marketplaceId),
            lender,
            paymentAmount
        );
```

If setLenderManager is called to change the lenderManager, in getLoanLender, since the bid.lender is not the current lenderManager, the old lenderManager address will be returned as the lender, and the repaid assets will be sent to the old lenderManager, resulting in the loss of the lender's assets

Impact

It may cause some Lenders to lose their assets

Code Snippet

https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L212-L229 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L560-L574 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L1037-L1047 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L744-L752

Tool used

Manual Review

Recommendation

Consider using MAGIC_NUMBER as bid.lender in claimLoanNFT and using that MAGIC_NUMBER in getLoanLender to do the comparison.

```
+   address MAGIC_NUMBER = 0x...;
    function claimLoanNFT(uint256 _bidId)
        external
        acceptedLoan(_bidId, "claimLoanNFT")
        whenNotPaused
    {
        // Retrieve bid
        Bid storage bid = bids[_bidId];

        address sender = _msgSenderForMarket(bid.marketplaceId);
        require(sender == bid.lender, "only lender can claim NFT");
        // mint an NFT with the lender manager
        lenderManager.registerLoan(_bidId, sender);
        // set lender address to the lender manager so we know to check the
 ↪  owner of the NFT for the true lender
-       bid.lender = address(lenderManager);
+       bid.lender = MAGIC_NUMBER;
    }
...
    function getLoanLender(uint256 _bidId)
        public
        view
        returns (address lender_)
    {
        lender_ = bids[_bidId].lender;

-       if (lender_ == address(lenderManager)) {
+       if (lender_ == MAGIC_NUMBER) {
```

```
            return lenderManager.ownerOf(_bidId);
        }
    }
```

Discussion

ethereumdegen

Github PR Issue 339 - Using magic number for lender manager control

IAm0x52

Fix looks good. Changed as recommended and is now using a magic number rather than the address of the lenderManager.

Issue M-13: A borrower/lender or liquidator will fail to withdraw the collateral assets due to reaching a gas limit

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/357

Found by

0xmuxyz, HonorLt, cccz, yixxas

Summary

Within the TellerV2#`submitBid()`, there is no limitation that how many collateral assets a borrower can assign into the `_collateralInfo` array parameter.

This lead to some bad scenarios like this due to reaching gas limit:

- A borrower or a lender fail to withdraw the collateral assets when the loan would not be liquidated.
- A liquidator will fail to withdraw the collateral assets when the loan would be liquidated.

Vulnerability Detail

Within the ICollateralEscrowV1, the `Collateral` struct would be defined line this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/interfaces/escrow/ICollateralEscrowV1.sol#L10-L15

```
struct Collateral {
    CollateralType _collateralType;
    uint256 _amount;
    uint256 _tokenId;
    address _collateralAddress;
}
```

Within the CollateralManager, the CollateralInfo struct would be defined like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L34-L37

```
/**
 * Since collateralInfo is mapped (address assetAddress => Collateral) that means
 * that only a single tokenId per nft per loan can be collateralized.
 * Ex. Two bored apes cannot be used as collateral for a single loan.
 */
struct CollateralInfo {
    EnumerableSetUpgradeable.AddressSet collateralAddresses;
```

```
    mapping(address => Collateral) collateralInfo;
}
```

Within the CollateralManager, the `_bidCollaterals` storage would be defined like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L27

```
// bidIds -> validated collateral info
mapping(uint256 => CollateralInfo) internal _bidCollaterals;
```

When a borrower submits a bid, the TellerV2#`submitBid()` would be called. Within the TellerV2#`submitBid()`, multiple collaterals, which are ERC20/ERC721/ERC1155, can be assigned into the `_collateralInfo` array parameter by a borrower. And then, these collateral assets stored into the `_collateralInfo` array would be associated with `bidId_` through internally calling the CollateralManager#`commitCollateral()` like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L311 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L325

```
function submitBid(
    address _lendingToken,
    uint256 _marketplaceId,
    uint256 _principal,
    uint32 _duration,
    uint16 _APR,
    string calldata _metadataURI,
    address _receiver,
    Collateral[] calldata _collateralInfo /// @audit
) public override whenNotPaused returns (uint256 bidId_) {
    ...
    bool validation = collateralManager.commitCollateral(
        bidId_,
        _collateralInfo /// @audit
    );
    ...
```

Within the CollateralManager#`commitCollateral()`, each collateral asset (`info`) would be associated with a `_bidId` respectively by calling the CollateralManager#`_commitCollateral()` in the for-loop like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L127

```
/**
 * @notice Checks the validity of a borrower's multiple collateral balances and
 ↪  commits it to a bid.
```

```
 * @param _bidId The id of the associated bid.
 * @param _collateralInfo Additional information about the collateral assets.
 * @return validation_ Boolean indicating if the collateral balances were
 ↪  validated.
 */
function commitCollateral(
    uint256 _bidId,
    Collateral[] calldata _collateralInfo  /// @audit
) public returns (bool validation_) {
    address borrower = tellerV2.getLoanBorrower(_bidId);
    (validation_, ) = checkBalances(borrower, _collateralInfo);

    if (validation_) {
        for (uint256 i; i < _collateralInfo.length; i++) {
            Collateral memory info = _collateralInfo[i];
            _commitCollateral(_bidId, info);  /// @audit
        }
    }
}
```

Within the CollateralManager#`_commitCollateral()`, the `_collateralInfo` would be stored into the `_bidCollaterals` storage like this:
https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L428
https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L430-L434

```
/**
 * @notice Checks the validity of a borrower's collateral balance and commits it
 ↪  to a bid.
 * @param _bidId The id of the associated bid.
 * @param _collateralInfo Additional information about the collateral asset.
 */
function _commitCollateral(
    uint256 _bidId,
    Collateral memory _collateralInfo
) internal virtual {
    CollateralInfo storage collateral = _bidCollaterals[_bidId];
    collateral.collateralAddresses.add(_collateralInfo._collateralAddress);
    collateral.collateralInfo[
        _collateralInfo._collateralAddress
    ] = _collateralInfo;  /// @audit
    ...
```

When the deposited-collateral would be withdrawn by a borrower or a lender, the CollateralManager#`withdraw()` would be called. Within the CollateralManager#`withdraw()`, the CollateralManager#`_withdraw()` would be

called like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L253 https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L255

```
/**
 * @notice Withdraws deposited collateral from the created escrow of a bid that
 ↪  has been successfully repaid.
 * @param _bidId The id of the bid to withdraw collateral for.
 */
function withdraw(uint256 _bidId) external {
    BidState bidState = tellerV2.getBidState(_bidId);
    if (bidState == BidState.PAID) {
        _withdraw(_bidId, tellerV2.getLoanBorrower(_bidId)); /// @audit
    } else if (tellerV2.isLoanDefaulted(_bidId)) {
        _withdraw(_bidId, tellerV2.getLoanLender(_bidId));  /// @audit
       ...
```

When the deposited-collateral would be liquidated by a liquidator, the CollateralManager#`liquidateCollateral()` would be called. Within the CollateralManager#`liquidateCollateral()`, the CollateralManager#`_withdraw()` would be called like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L278

```
/**
 * @notice Sends the deposited collateral to a liquidator of a bid.
 * @notice Can only be called by the protocol.
 * @param _bidId The id of the liquidated bid.
 * @param _liquidatorAddress The address of the liquidator to send the
 ↪  collateral to.
 */
function liquidateCollateral(uint256 _bidId, address _liquidatorAddress)
    external
    onlyTellerV2
{
    if (isBidCollateralBacked(_bidId)) {
        BidState bidState = tellerV2.getBidState(_bidId);
        require(
            bidState == BidState.LIQUIDATED,
            "Loan has not been liquidated"
        );
        _withdraw(_bidId, _liquidatorAddress);  /// @audit
    }
}
```

Within the CollateralManager#`_withdraw()`, each collateral asset (`collateralInfo._collateralAddress`) would be withdrawn by internally calling the

ICollateralEscrowV1#`withdraw()` in a for-loop like this: https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L394-L409

```
/**
 * @notice Withdraws collateral to a given receiver's address.
 * @param _bidId The id of the bid to withdraw collateral for.
 * @param _receiver The address to withdraw the collateral to.
 */
function _withdraw(uint256 _bidId, address _receiver) internal virtual {
    for (
        uint256 i;
        i < _bidCollaterals[_bidId].collateralAddresses.length(); /// @audit
        i++
    ) {
        // Get collateral info
        Collateral storage collateralInfo = _bidCollaterals[_bidId]
            .collateralInfo[
                _bidCollaterals[_bidId].collateralAddresses.at(i)
            ];
        // Withdraw collateral from escrow and send it to bid lender
        ICollateralEscrowV1(_escrows[_bidId]).withdraw(   /// @audit
            collateralInfo._collateralAddress,
            collateralInfo._amount,
            _receiver
        );
```

However, within the TellerV2#`submitBid()`, there is no limitation that how many collateral assets a borrower can assign into the `_collateralInfo` array parameter.

This lead to a bad scenario like below:

- A borrower assign too many number of the collateral assets (ERC20/ERC721/ERC1155) into the `_collateralInfo` array parameter when the borrower call the TellerV2#`submitBid()` to submit a bid.
- Then, a lender accepts the bid via calling the TellerV2#`lenderAcceptBid()`
- Then, a borrower or a lender try to withdraw the collateral, which is not liquidated, by calling the CollateralManager#`withdraw()`. Or, a liquidator try to withdraw the collateral, which is liquidated, by calling the CollateralManager#`liquidateCollateral()`
- But, the transaction of the CollateralManager#`withdraw()` or the CollateralManager#`liquidateCollateral()` will be reverted in the for-loop of the CollateralManager#`_withdraw()` because that transaction will reach a gas limit.

Impact

Due to reaching gas limit, some bad scenarios would occur like this:

- A borrower or a lender fail to withdraw the collateral assets when the loan would not be liquidated.
- A liquidator will fail to withdraw the collateral assets when the loan would be liquidated.

Code Snippet

- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/interfaces/escrow/ICollateralEscrowV1.sol#L10-L15
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L34-L37
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L27
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L311
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/TellerV2.sol#L325
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L127
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L428
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L430-L434
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L253
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L255
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L278
- https://github.com/sherlock-audit/2023-03-teller/blob/main/teller-protocol-v2/packages/contracts/contracts/CollateralManager.sol#L394-L409

Tool used

Manual Review

Recommendation

Within the TellerV2#`submitBid()`, consider adding a limitation about how many collateral assets a borrower can assign into the `_collateralInfo` array parameter.

Discussion

ethereumdegen

Thank you for your feedback. This is very similar / essentially the same as the 'collateral poisoning' issue that had been identified in the README of this contest as a known-issue: it had been explained and known that collateral could be made impossible to withdraw which could impact the ability to do the last repayment of a loan. This is a slight variation in that it describes that the collateral could be so vast that withdrawing it would exceed the gas limit of a block. Thank you for this perspective. In any case we do plan to separate the repayment logic from the collateral withdraw logic to mitigate such an issue.

ctf-sec

Escalate for 10 USDC. this is low / medium issue.

As the sponsor say

> his is very similar / essentially the same as the 'collateral poisoning' issue that had been identified in the README of this contest as a known-issue: it had been explained and known that collateral could be made impossible to withdraw which could impact the ability to do the last repayment of a loan.

according to the previous description, the issue should be marked as low and non-reward

but DOS and exceed the gas limit of block make this a medium, definitely not a high finding

According to https://docs.sherlock.xyz/audits/judging/judging

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

and

> Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue? It would not count if the DOS, etc. lasts a

> known, finite amount of time <1 year. If it will result in funds being inaccessible for >=1 year, then it would count as a loss of funds and be eligible for a Medium or High designation. The greater the cost of the attack for an attacker, the less severe the issue becomes.

The cost of making the collateral loop for exceed block gas limit is clearly very high + it only impact single lendering / borrow loan, not all loan states.

So this should be a medium / low finiding, definitely not high severity issue.

sherlock-admin

> Escalate for 10 USDC. this is low / medium issue.
>
> As the sponsor say
>
> > his is very similar / essentially the same as the 'collateral poisoning' issue that had been identified in the README of this contest as a known-issue: it had been explained and known that collateral could be made impossible to withdraw which could impact the ability to do the last repayment of a loan.
>
> according to the previous description, the issue should be marked as low and non-reward
>
> but DOS and exceed the gas limit of block make this a medium, definitely not a high finding
>
> According to https://docs.sherlock.xyz/audits/judging/judging
>
> > Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> and
>
> > Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue? It would not count if the DOS, etc. lasts a known, finite amount of time <1 year. If it will result in funds being inaccessible for >=1 year, then it would count as a loss of funds and be eligible for a Medium or High designation. The greater the cost of the attack for an attacker, the less severe the issue becomes.
>
> The cost of making the collateral loop for exceed block gas limit is clearly

> very high + it only impact single lendering / borrow loan, not all loan states.
>
> So this should be a medium / low finiding, definitely not high severity issue.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

Trumpero

This issue differs from the known issue as it highlights a scenario where the loan was successfully accepted but can't be withdrawn due to the gas limit. However, this situation is unlikely, so I agree that it should be a medium issue.

hrishibhat

Escalation accepted

Valid medium This can be considered as a valid medium.

sherlock-admin

> Escalation accepted
>
> Valid medium This can be considered as a valid medium.

```
This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on
↪  this issue.
```

IAm0x52

Fixed here by separating withdraw and repay logic

Issue M-14: Premature Liquidation When a Borrower Pays early

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/494

Found by

branch_indigo

Summary

On TellerV2 markets, whenever a borrower pays early in one payment cycle, they could be at risk to be liquidated in the next payment cycle. And this is due to a vulnerability in the liquidation logic implemented in `_canLiquidateLoan`. Note: This issue is submitted separately from issue #2 because the exploit is based on user behaviors regardless of a specific market setting. And the vulnerability might warrant a change in the liquidation logic.

Vulnerability Detail

In TellerV2.sol, the sole liquidation logic is dependent on the time gap between now and the previous payment timestamp. But a user might decide to pay at any time within a given payment cycle, which makes the time gap unreliable and effectively renders this logic vulnerable to exploitation.

```
return (uint32(block.timestamp) -
    _liquidationDelay -
    lastRepaidTimestamp(_bidId) >
    bidDefaultDuration[_bidId]);
```

Suppose a scenario where a user takes on a loan on a market with 3 days payment cycle and 3 days paymentDefaultDuration. And the loan is 14 days in duration. The user decided to make the first minimal payment an hour after receiving the loan, and the next payment due date is after the sixth day. Now 5 days passed since the user made the first payment, and a liquidator comes in and liquidates the loan and claims the collateral before the second payment is due.

Here is a test to show proof of concept for this scenario.

Impact

Given the fact that this vulnerability is not market specific and that users can pay freely during a payment cycle, it's quite easy for a liquidator to liquidate loans prematurely. And the effect might be across multiple markets.

When there are proportional collaterals, the exploit can be low cost. An attacker could take on flash loans to pay off the principal and interest, and the interest could be low when early in the loan duration. The attacker would then sell the collateral received in the same transaction to pay off flash loans and walk away with profits.

Code Snippet

https://github.com/teller-protocol/teller-protocol-v2/blob/cb66c9e348cdf1fd6d9b0416a49d663f5b6a693c/packages/contracts/contracts/TellerV2.sol#L965-L969

Tool used

Manual Review

Recommendation

Consider using the current timestamp - previous payment due date instead of just `lastRepaidTimestamp` in the liquidation check logic. Also, add the check to see whether a user is late on a payment in `_canLiquidateLoan`.

Discussion

ethereumdegen

In this logic, we should be calculating the default date(s) based relative to a 'due date' and not a 'repaid date' . We will look into this more closely.

ethereumdegen

Github PR: Issue 494 - improving logic for is loan liquidateable

IAm0x52

Fix looks good. Default date is now calculated from due date rather than repaid date

Issue M-15: A malicious market owner/protocol owner can front-run calls to lenderAcceptBid and change the marketplace fee to steal lender funds

Source: https://github.com/sherlock-audit/2023-03-teller-judging/issues/497

Found by

0xGoodess, BAHOZ, Bauer, HonorLt, MiloTruck, Nyx, Saeedalipoor01988, cducrest-brainbot, ck, ctf_sec, deadrxsezzz, dingo, duc, hake, immeas, innertia, jpserrat, monrel, spyrosonic10, tallo, whoismatthewmc1

Lines of Code

https://github.com/teller-protocol/teller-protocol-v2/blob/develop/packages/contracts/contracts/TellerV2.sol#L470 https://github.com/teller-protocol/teller-protocol-v2/blob/develop/packages/contracts/contracts/ProtocolFee.sol#L44 https://github.com/teller-protocol/teller-protocol-v2/blob/develop/packages/contracts/contracts/MarketRegistry.sol#L621

Summary

Malicious market owners and protocol owners can arbitrary set fees to extraordinary rates to steal all of the lenders funds.

Vulnerability Detail

A malicious market owner can front run lenders who wish to accept a bid through `lenderAcceptBid` by calling `MarketRegistry.setMarketFeePercent` to set the marketplace fee to 100%. This allows the malicious market owner to steal 100% of the funds from the lender. The same thing can be done by a malicious protocol owner by calling `ProtocolFee.setProtocolFee`

Impact

Lender loses all their funds on a bid they accept due to malicious or compromised market owner/protocol owner.

Code Snippet

```
function lenderAcceptBid(uint256 _bidId)
        external
        override
```

```
        pendingBid(_bidId, "lenderAcceptBid")
        whenNotPaused
        returns (
            uint256 amountToProtocol,
            uint256 amountToMarketplace,
            uint256 amountToBorrower
        )
{
        //..

        //@audit here the fee amounts are calculated
        amountToProtocol = bid.loanDetails.principal.percent(protocolFee());

        //@audit this value is what is front-ran by the marketplace
↪   owner/protocol owner through MarketRegistry.setMarketFeePercent
        amountToMarketplace = bid.loanDetails.principal.percent(
            marketRegistry.getMarketplaceFee(bid.marketplaceId)
        );

        //@audit here the total amount to send to the borrower is calculated by
↪   subtracting the fees
        //from the principal value.
        amountToBorrower =
            bid.loanDetails.principal -
            amountToProtocol -
            amountToMarketplace;

        //@audit transfer fee to protocol
        bid.loanDetails.lendingToken.safeTransferFrom(
            sender,
            owner(),
            amountToProtocol
        );

        //@audit transfer fee to marketplace
        bid.loanDetails.lendingToken.safeTransferFrom(
            sender,
            marketRegistry.getMarketFeeRecipient(bid.marketplaceId),
            amountToMarketplace
        );
        //..
}
```

```
function setMarketFeePercent(uint256 _marketId, uint16 _newPercent)
    public
    ownsMarket(_marketId)
{
```

```
    require(_newPercent >= 0 && _newPercent <= 10000, "invalid percent");
    if (_newPercent != markets[_marketId].marketplaceFeePercent) {
        //@audit here the market fee is set
        markets[_marketId].marketplaceFeePercent = _newPercent;
        emit SetMarketFee(_marketId, _newPercent);
    }
}
```

Tool used

Manual Review

Recommendation

1. Add a timelock delay for setMarketFeePercent/setProtocolFee
2. allow lenders to specify the exact fees they were expecting as a parameter to `lenderAcceptBid` Note: The developers seem to be aware of this attack vector but their doesn't appear to be a fix in this case

"Market owners should NOT be able to race-condition attack borrowers or lenders by changing market settings while bids are being submitted or accepted (while tx are in mempool). Care has been taken to ensure that this is not possible (similar in theory to sandwich attacking but worse as if possible it could cause unexpected and non-consentual interest rate on a loan) and further-auditing of this is welcome. The best way to defend against this is to allow borrowers and lenders to specify such loan parameters in their TX such that they are explicitly consenting to them in the tx and then reverting if the market settings conflict with those tx arguments."

Discussion

securitygrid

Escalate for 10 USDC This is not valid M. Accroding to README.md: Market owners should NOT be able to race-condition attack borrowers or lenders by changing market settings while bids are being submitted or accepted

sherlock-admin

> Escalate for 10 USDC This is not valid M. Accroding to README.md: Market owners should NOT be able to race-condition attack borrowers or lenders by changing market settings while bids are being submitted or accepted

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

ethereumdegen

Github PR: https://github.com/teller-protocol/teller-protocol-v2/pull/81

hrishibhat

Escalation rejected

Valid medium This is a valid issue and protocol expected this to be audited and is mentioned in the readme

sherlock-admin

> Escalation rejected
>
> Valid medium This is a valid issue and protocol expected this to be audited and is mentioned in the readme

```
This issue's escalations have been rejected!

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

IAm0x52

Fix looks good. LenderAcceptBid now allows users to specify a max fee