CANTINA

Particle LAMM

Competition

March 28, 2024

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 `getOwedFee` can incorrectly return zero because of fee growth underflow . . . 4
  - 3.1.2 `openPosition` should not charge fees on marginfrom or charge fees on `marginTo` as well . . . 5
  - 3.1.3 Blast is configured in the implementation contract constructor, not the proxy . . . 6
  - 3.1.4 `particlepositionmanager` fails to implement `pausableupgradeable` properly . . . 7
  - 3.1.5 `swap` and `addpremium` are susceptible to bad partial trading . . . 7
- 3.2 Medium Risk . . . 8
  - 3.2.1 When owner reduces loan_term, it may cause the borrower to be liquidated immediately . . . 8
  - 3.2.2 If the Blast admin calls `adminClaimGas` on `particlePositionManager`, the latter won't be able to claim gas anymore . . . 9
  - 3.2.3 There's no way to transfer out native balance from the particlepositionmanager contract . . . 10
  - 3.2.4 While ‘particlePositionManager contract is paused, positions could go liquidatable . . . 11
  - 3.2.5 Liquidity can still be supplied when the protocol is paused, even though `mint` is disabled 12
  - 3.2.6 `USDB` and `WETH` addresses are misconfigured in `blast.sol` . . . 12
  - 3.2.7 Premature liquidation in `liquidateposition()` due to improper `isliquidatable` validation . . . 12

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Particle builds a permissionless, oracle-free protocol to enable leverage trading for any ERC20 tokens. Traders can long/short any token with leverage even at the token launch time. LPs can boost liquidity efficiency and earn higher yield.

From Feb 19th to Feb 29th Cantina hosted a competition based on particle-erc20. The participants identified a total of **24** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 5
- Medium Risk: 7
- Low Risk: 6
- Gas Optimizations: 1
- Informational: 5

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 `getOwedFee` can incorrectly return zero because of fee growth underflow

Submitted by 100proof

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description

Although function `getFeeGrowthInside` correctly accounts for Uniswap V3's principled use of underflow to calculate `feeGrowthInside[0/1]X128`, function `getOwedFee` will incorrectly returns zero for `token[0/1]Owed` in the following situation:

- A Uniswap position has its `feeGrowthInside0X128` calculated, but an underflow happened during the calculation. e.g. `feeGrowthInside0X128 == type(uint256).max - 1e18 + 1`.
- Particle user calls `openPosition` and has this value stored in in the `Lien.Info`'s `feeGrowthInside0LastX128` field.
- While the Particle position is open, the Uniswap position has its values changed such that a call to `Base- getFeeGrowthInside` returns a small positive value. e.g. `feeGrowthInside0X128 == 2e18`.
- Particle user calls `closePosition` calling `getOwedFee`. However, it is not true that `feeGrowthInside0X128 > feeGrowthInside0LastX128` on `Base.sol- L378` so `token0Owed` is returned as zero.

If we look at Uniswap V3's Position- update function we have:

```
uint128 tokensOwed0 =
    uint128(
        FullMath.mulDiv(
            feeGrowthInside0X128 - _self.feeGrowthInside0LastX128,
            _self.liquidity,
            FixedPoint128.Q128
        )
    );
uint128 tokensOwed1 =
    uint128(
        FullMath.mulDiv(
            feeGrowthInside1X128 - _self.feeGrowthInside1LastX128,
            _self.liquidity,
            FixedPoint128.Q128
        )
    );
```

Note that there is no check that `freeGrowthInside0X128 > _self.feeGrowthInside0LastX128`.

If we use the values from our example, `2e18 - (type(uint256).max - 1e18 + 1)` will evaluate to `3e18` because Uniswap V3 contracts are compiled with Solidity version `< 8` where underflow is permitted. The impact is that `getOwedFee` returns zero when it shouldn't and LPs lose out on fees.

Note: the same problem occurs in `ParticleInfoReader- getInterestInfo` but impact is lower.

More information can be found in these GitHub issues:

- Uniswap.
- Jeiwan's Uniswap V3 book.

Recommendation: Rewrite `getOwedFee` as:

```
function getOwedFee(
    uint256 feeGrowthInside0X128,
    uint256 feeGrowthInside1X128,
    uint256 feeGrowthInside0LastX128,
    uint256 feeGrowthInside1LastX128,
    uint128 liquidity
) internal pure returns (uint128 token0Owed, uint128 token1Owed) {
    unchecked {
        if (feeGrowthInside0X128 != feeGrowthInside0LastX128) {
            token0Owed = uint128(
                FullMath.mulDiv(feeGrowthInside0X128 - feeGrowthInside0LastX128, liquidity, FixedPoint128.Q128)
            );
        }
        if (feeGrowthInside1X128 != feeGrowthInside1LastX128) {
            token1Owed = uint128(
                FullMath.mulDiv(feeGrowthInside1X128 - feeGrowthInside1LastX128, liquidity, FixedPoint128.Q128)
            );
        }
    }
}
```

Similarly for `ParticleInfoReader.getInterestInfo`.

Particle: Fixed. The `< 0.8.0` underflow is now using `unchecked`.

3.1.2 `openPosition` should not charge fees on marginfrom or charge fees on `marginTo` as well

Submitted by cccz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In `openPosition`, `marginFrom + amountFromBorrowed` is used as the base for calculating the fee:

```
if (cache.feeFactor > 0) {
    cache.feeAmount = ((params.marginFrom + cache.amountFromBorrowed) * cache.feeFactor) / Base.BASIS_POINT;
```

However there are two reasons why `marginFrom` should not be fee charged- First of all, `marginFrom` is provided by the user and not borrowed from somewhere else, so `marginFrom` should not be fee charged.

Most importantly, the user can bypass the `marginFrom` fee by supplying more `marginTo` to make `marginFrom` zero. For example, consider ETH:USDC = 3000, to make long ETH, the user needs to provide a marginFrom of 6000 USDC. However, since the protocol will charge fees on `marginFrom`, the user can first exchange the 6000 USDC to 2 ETH externally, and then provide 2 ETH as the `marginTo`, thus avoiding the protocol to charge fees on the `marginFrom`. This actually overcharges most users.

Recommendation:

It is recommended to not charge fees on marginFrom in openPosition.

```
- cache.feeAmount = ((params.marginFrom + cache.amountFromBorrowed) * cache.feeFactor) / Base.BASIS_POINT;
+ cache.feeAmount = (cache.amountFromBorrowed * cache.feeFactor) / Base.BASIS_POINT;
```

Or to charge fees on `marginTo` as well.

Particle: Mitigated. It's our protocol's design for applying the position fee to which part of the principal. Also, protected by the server side signature, borrower can't just overwrite the frontend to use `marginTo` instead of `marginFrom`. Hence this significantly reduces the severity and we keep this `feeAmount` calculation for consistency.

3.1.3 Blast is configured in the implementation contract constructor, not the proxy

Submitted by neumo, also found by Jonatas Martins and Sujith Somraaj

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `ParticlePositionManager` contract has a line in the constructor (which is currently commented out to deploy on mainnet, but will be uncommented when it is deployed on Blast) intended to configure the yield and gas modes of the contract as `claimable`.

ParticlePositionManager:

```
// ...
constructor() payable {
    _disableInitializers();
    // Blast.configure();
}
// ...
```

Blast (library):

```
// ...
function configure() external {
    IBlast(BLAST).configureClaimableYield();
    IBlast(BLAST).configureClaimableGas();
    IERC20Rebasing(WETHB).configure(YieldMode.CLAIMABLE);
    IERC20Rebasing(USDB).configure(YieldMode.CLAIMABLE);
}
// ...
```

The `ParticlePositionManager` is supposed to be sitting behind a proxy, so configuring yield modes in the constructor only affects the implementation, not the proxy. The balances of `Blast ETH`, `USDB` and `WETHB` will be accruing yield in the proxy contract, not the implementation. And since the proxy will have Disabled yield mode (the default for newly created contracts in Blast), they will accrue zero yield and there's no way to change the modes in the current codebase.

Regarding gas, the contract accruing gas will be the proxy contract (not the implementation) and since the Gas mode will be Void (the default mode), and no mechanism in place to change that, it will accrue zero gas fees.

Recommendation: The configuration of the Blast yield modes should be done in the `initialize` function, which should be called through the proxy contract.

```
  constructor() payable {
      _disableInitializers();
-     // Blast.configure();
  }
  // ...

  function initialize(
      uint256 feeFactor,
      uint128 liquidationRewardFactor,
      uint256 loanTerm,
      uint256 treasuryRate,
      address blastAdmin
  ) external initializer {
      __UUPSUpgradeable_init();
      __Ownable_init();
      __Pausable_init();
      updateFeeFactor(feeFactor);
      updateLiquidationRewardFactor(liquidationRewardFactor);
      updateLoanTerm(loanTerm);
      updateTreasuryRate(treasuryRate);
      updateBlastPointsAdmin(blastAdmin);
+     // Blast.configure();
  }
```

Particle: Fixed. We moved it to proxy.

3.1.4 `particlepositionmanager` fails to implement `pausableupgradeable` properly

Submitted by Sujith Somraaj, also found by neumo, gmhacker, m4rio and David Ch.

Severity: High Risk

Context: ParticlePositionManager.sol#L27

Description: The contract `ParticlePositionManager` inherits the `PausableUpgradeable` contract from the Openzeppelin library and uses the `whenNotPaused` modifier across multiple instances in the contract.

However, the contract fails to implement the actual ability to pause the contract. Due to which, the import gives a false sense of security where the contract can be actually paused in emergency scenario but it cannot.

This issue is classified as HIGH as it can lead to unexpected protocol behavior and the ability to not pause the protocol under critical attacks.

Recommendation: Override the internal `_pause()` and `_unpause` methods in the inherited contract and implement the pause functionality completely. For more info, please read the OZ docs here, where the exact impact is added in the note.

```
+ function pause() onlyOwner {
+ _pause()
+ }

+ function unpause() onlyOwner {
+ _unpause()
+ }
```

Particle: Fixed. We don't use pausable anymore.

3.1.5 `swap` and `addpremium` are susceptible to bad partial trading

Submitted by David Ch., also found by cccz, gmhacker and Sujith Somraaj

Severity: High Risk

Context: Lien.sol#L120

Description: `amountOutMinimum` is used to specify the minimum amount of tokens the caller wants to be returned from a swap. Using `amountOutMinimum = 0` tells the swap that the caller will accept a minimum amount of 0 output tokens from the swap, opening up the user to a catastrophic loss of funds in case of unexpected frontruns, bad market situations, etc... This in general had been exploited via MEV bot sandwich attacks.

Certainly `SqrtLimitX96` generates some sort of protection, notice first the difference between the two protections:

- `minOutputAmount`: all or nothing, if you don't give this amount, bad, let's go revert.
- `SqrtLimitX96`: related to price impact, until here, I allow you to trade as much tokens as posibles. ***Note**: allows partial trading*.

A well explained post can be found here.

That means, that even if I can receive a `minOutputAmount` of 1500 of `token_example` under a certain price impact, if there is only 1 `token_example` available on that price impact, I will take the transaction. What is good for many cases.

The thing is, I transfer to `ParticlePositionManager` contract assets to swap for those 1500 tokens, but there is only 1 I receive because of liquidity, what indeed, can be farmed by MEV to be within the highest price posible. This partial trading would make sense if I were trading directly with uniswap v3 from my EOA, as I would hold the non traded amount.

Now this scenario in `ParticlePositionManager`:

```
/// @inheritdoc IParticlePositionManager
function swap(
    address tokenFrom,
    address tokenTo,
    uint24 fee,
    uint256 amountFrom,
    uint160 sqrtPriceLimitX96
) external override nonReentrant whenNotPaused returns (uint256 amountReceived) {
    //@audit I transfer here the whole amount for my 1500 tokens
    TransferHelper.safeTransferFrom(tokenFrom, msg.sender, address(this), amountFrom);
    //@audit trade under a certain price impact but receive 1 token
    amountReceived = Base.swapExactIn(tokenFrom, tokenTo, fee, amountFrom, 0, sqrtPriceLimitX96);
    //@audit now transfer 1 token for the price of 1500 tokens to me and the contract holds the rest of the
↪   value of my transfer.
    TransferHelper.safeTransfer(tokenTo, msg.sender, amountReceived);
}
```

Basically, it is not desirable for users to allow partial trading in this context, as the remaining value would be hold by the contract and not returned to the user.

This issue of not allowing to set `minOutAmount` also happens in 2 other functions in `ParticlePositionManager`, in theory, with some checks regarding the received amount: `openPosition` and `_closePosition`

Recommendation: The recommendations to avoid losing funds are:

- Include a minOutputAmount as a param, so in case of an issue is user's fault and not protocol + market conditions / MEV attacks that affect the value received.
- Do a check in balance prev and post trade and return excess amount to the sender.

Particle: Fixed. We've used `minimumAmount` as opposed to `sqrtPriceLimitX96`.

3.2 Medium Risk

3.2.1 When owner reduces loan_term, it may cause the borrower to be liquidated immediately

Submitted by cccz

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When `block.timestamp > cutoffTime + LOAN_TERM`, the borrower will be liquidated:

```
function isLiquidatable(
    uint128 tokenFromPremium,
    uint128 tokenToPremium,
    uint128 tokenFromOwed,
    uint128 tokenToOwed,
    uint40 tokenId
) public view returns (bool) {
    ///@dev the liquidation condition is that
    ///     EITHER (premium is not enough) OR (cutOffTime > 0 AND currentTime > cutOffTime + LOAN_TERM)
    uint32 cutoffTime = lps.getRenewalCutoffTime(tokenId);
    return
        (tokenFromPremium < tokenFromOwed || tokenToPremium < tokenToOwed) ||
        (cutoffTime > 0 && cutoffTime + LOAN_TERM < block.timestamp);
}
```

The problem here is that the owner may call `updateLoanTerm` to reduce the `LOAN_TERM`, which may cause the borrower to be immediately liquidated:

```
function updateLoanTerm(uint256 loanTerm) public override onlyOwner whenNotPaused {
    if (loanTerm > _LOAN_TERM_MAX) revert Errors.InvalidValue();
    LOAN_TERM = loanTerm;
    emit UpdateLoanTerm(loanTerm);
}
```

Considering that the current `LOAN_TERM` is 7 days, the NFT owner calls `reclaimLiquidity` and the borrower must close the position within 7 days, otherwise it will be liquidated.

The borrower plans to close the position on the 5th day, but the owner calls `updateLoanTerm` to reduce `LOAN_TERM` to 3 days, at which time the borrower will be liquidated immediately.

Recommendation: It is recommended to store `block.timestamp + LOAN_TERM` instead of `block.timestamp` in `updateRenewalCutoffTime`:

```
  function updateRenewalCutoffTime(
      mapping(uint256 => Info) storage self,
      uint256 tokenId
  ) internal returns (uint32 cutOffTime) {
      Info storage info = self[tokenId];
      cutOffTime = uint32(block.timestamp);
-     info.renewalCutoffTime = cutOffTime;
+     info.renewalCutoffTime = cutOffTime + LOAN_TERM;
  }
  // ...
  function isLiquidatable(
      uint128 tokenFromPremium,
      uint128 tokenToPremium,
      uint128 tokenFromOwed,
      uint128 tokenToOwed,
      uint40 tokenId
  ) public view returns (bool) {
      ///@dev the liquidation condition is that
      ///     EITHER (premium is not enough) OR (cutOffTime > 0 AND currentTime > cutOffTime + LOAN_TERM)
      uint32 cutoffTime = lps.getRenewalCutoffTime(tokenId);
      return
          (tokenFromPremium < tokenFromOwed || tokenToPremium < tokenToOwed) ||
-         (cutoffTime > 0 && cutoffTime + LOAN_TERM < block.timestamp);
+         (cutoffTime > 0 && cutoffTime < block.timestamp);
  }
```

Particle: Acknowledged. `LOAN_TERM` shouldn't be updated often (initially by multisign, in the future by governance). We store timestamp for simplicity.

3.2.2 If the Blast admin calls `adminClaimGas` on `particlePositionManager`, the latter won't be able to claim gas anymore

Submitted by neumo

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Blast contract has a function called `adminClaimGas`, only callable by Blast's admin (see Gas.sol#L128-L133).

```
// ...
function adminClaimGas(address contractAddress) external onlyAdmin returns (uint256) {
    (, uint256 etherBalance,,) = readGasParams(contractAddress);
    _updateGasParams(contractAddress, 0, 0, GasMode.VOID);
    SafeTransferLib.safeTransferETH(blastFeeVault, etherBalance);
    return etherBalance;
}
// ...
```

Once called, it transfers all the gas accrued by a contract to the `blastFeeVault`, and sets the gas params to 0 balance, 0 seconds and changes the Gas mode to Void.

That means if the admin calls the function for the `ParticlePositionManager` it will stop accruing gas rewards, and will lose all future gas related income from that point, until the contract is upgraded to be able to change the gas mode again.

Recommendation: There should be a function in the `ParticlePositionManager` to allow the contract to set the Gas mode to Claimable (to be put in the Blast library):

```
// ...
function setGasMode() external {
    IBlast(BLAST).configureClaimableGas();
}
```

Particle: Fixed. Added `setGasMode()`.

3.2.3 There's no way to transfer out native balance from the particlepositionmanager contract

Submitted by neumo

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `ParticlePositionManager` contract is intended to be deployed in the `Blast` network. `Blast` allows contracts and EOAs to set their Yield mode. As seen below, the contract configures itself at construction time

```
contract ParticlePositionManager
// ...
    constructor() payable {
        _disableInitializers();
        // Blast.configure();
    }
// ...
```

And the Blast library sets the claim mode and gas mode of the contract as Claimable and the yield mode of both the `USDB` and `WETHB` ERC20 rebasing tokens as Claimable:

```
library Blast
// ...
    function configure() external {
        IBlast(BLAST).configureClaimableYield();
        IBlast(BLAST).configureClaimableGas();
        IERC20Rebasing(WETHB).configure(YieldMode.CLAIMABLE);
        IERC20Rebasing(USDB).configure(YieldMode.CLAIMABLE);
    }
// ...
```

The owner of the `ParticlePositionManager` contract will have the power of calling function `claimYield-MaxGas` .

```
function claimYieldMaxGas(
    address recipient,
    uint256 ethToClaim,
    uint256 wethToClaim,
    uint256 usdbToClaim
)
    external
    override
    onlyOwner
    nonReentrant
    whenNotPaused
    returns (uint256 eth, uint256 weth, uint256 usdb, uint256 gas)
{
    (eth, weth, usdb, gas) = Blast.claimYieldMaxGas(recipient, address(this), ethToClaim, wethToClaim,
↪   usdbToClaim);
    emit ClaimYieldGas(recipient, eth, weth, usdb, gas);
}
```

Which ends up calling `claimYieldMaxGas` in the `Blast` library:

```
function claimYieldMaxGas(
    address recipient,
    address protocol,
    uint256 ethToClaim,
    uint256 wethToClaim,
    uint256 usdbToClaim
) external returns (uint256 eth, uint256 weth, uint256 usdb, uint256 gas) {
    eth = IBlast(BLAST).claimYield(protocol, recipient, ethToClaim);
    weth = IERC20Rebasing(WETHB).claim(recipient, wethToClaim);
    usdb = IERC20Rebasing(USDB).claim(recipient, usdbToClaim);
    gas = IBlast(BLAST).claimMaxGas(protocol, recipient);
}
```

This way, the owner can collect the yields generated in Blast's native ETH, `USDB` and `WETHB` and also the rewards for the gas used by the contract. To generate Blast's native ETH yield, the contract must have a positive balance (at least 1 share) and this can happen for instance if someone bridges ETH from mainnet

to Blast at the `ParticlePositionManager` contract address. This positive balance will generate yield which the owner can claim later.

The issue here is that the contract does not have any mechanism to transfer this Blast's ETH balance out, or bridge it back to mainnet, meaning it will get locked until the contract is upgraded to allow it.

The fact that the contract is upgradeable is what has made me evaluate this issue as medium instead of high.

Recommendation: The contract could implement a function that would allow the owner to transfer out native balance. Something like this:

```
function adminTransferETH(address receiver, uint256 amount) external onlyOwner {
    SafeTransferLib.safeTransferETH(receiver, amount);
}
```

Particle: Acknowledged. As designed, Particle contract shouldn't hold any native ETH, since uniswap v3 standard only supports ERC20 tokens. So Particle contract doesn't need to set the native ETH yield mode.

3.2.4 While 'particlePositionManager contract is paused, positions could go liquidatable

Submitted by neumo, also found by gmhacker

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Contract `ParticlePositionManager` uses modifier `whenNotPaused` from inherited contract `PausableUpgradeable` to disallow calling certain functions if the contract is paused. The following functions, among others, use this modifier:

- `closePosition(DataStruct.ClosePositionParams calldata params)`
- `addPremium(DataStruct.AddPremiumParams calldata params)`

This way, borrowers with open positions for which the LP owner has called `reclaimLiquidity` and started the three day period, could see the period end before the contract is unpaused, making their positions liquidatable without having had the chance of closing their positions. Also, positions could go liquidatable while contract is paused due to not enough premium, and the borrower will not be able to add premium before it's unpaused.

This would make the moment the contract is unpaused an unfair moment for some borrowers, who would see their positions liquidated.

Recommendation: I'm unsure of wether it's a good idea to allow closing positions or adding premium while the contract is paused. What could be done would be to add a storage variable to set when the contract is unpaused to give time to borrowers to handle their positions before they are liquidatable. Something like this:

```
uint64 public minLiquidationTimestamp;
uint64 constant LIQUIDATION_UNPAUSE_GRACE_PERIOD = 6 hours;
// ...
function unpause() public onlyOwner {
    minLiquidationTimestamp = block.timestamp + LIQUIDATION_UNPAUSE_GRACE_PERIOD;
    _unpause();
}
// ...
function liquidatePosition(
    DataStruct.ClosePositionParams calldata params,
    address borrower
) external override nonReentrant whenNotPaused {
    require(block.timestamp >= minLiquidationTimestamp);
// ...
```

Particle: Fixed. We don't use pausable pattern any more (instead we use server side signature).

3.2.5 Liquidity can still be supplied when the protocol is paused, even though `mint` is disabled

Submitted by gmhacker, also found by 100proof and m4rio

Severity: Medium Risk

Context: ParticlePositionManager.sol#L107-L120

Description: The `onERC721Received` function is used to allow a direct transfer of a UniswapV3 position NFT to `ParticlePositionManager`, using `ERC721.safeTransferFrom`. Alternatively, a user can call `ParticlePositionManager.mint`, where tokens will be transferred to the contract and the Particle manager will mint the NFT in Uniswap's position manager.

When the protocol gets paused, the `mint` function cannot be used any longer. That's because the `mint` function uses the `whenNotPaused` modifier. However, it's still possible to supply liquidity by using a direct NFT transfer, given that `onERC721Received` is not checking whether or not the protocol is paused. This means that users can actually still provide liquidity to the Particle position manager in an emergency situation where the protocol is paused. Importantly, the user will not be able to remove their liquidity because those functions are paused, therefore a user can supply liquidity to `ParticlePositionManager` without realising that the protocol is paused, and end up not being able to remove the liquidity, at least while the protocol is paused.

Recommendation: Consider adding the `whenNotPaused` modifier to `onERC721Received` as well, avoiding confusing situations for the users.

Particle: Fixed. We don't use pausable pattern any more.

3.2.6 `USDB` and `WETH` addresses are misconfigured in `blast.sol`

Submitted by Sujith Somraaj, also found by Jonatas Martins and David Ch.

Severity: Medium Risk

Context: Blast.sol#L8-L9

Description: The addresses of `USDB` and `WETH` are constants in the `Blast.sol` contract file; however, these variables are declared wrongly.

As per docs, the addresses of USDB and WETH are `0x4300000000000000000000000000000000000003` and `0x4300000000000000000000000000000000000004` respectively.

This value can also be seen in the precompiles of the blast node code. The impact would be unexpected protocol behavior, which is not ideal for integrators, thus classifying it as a medium issue.

Recommendation: Update the correct address for `USDC` and `WETH` accordingly.

```
- address constant WETHB = 0x4200000000000000000000000000000000000022;
- address constant USDB = 0x4200000000000000000000000000000000000023;

+ address constant WETHB = 0x4300000000000000000000000000000000000004;
+ address constant USDB = 0x4300000000000000000000000000000000000003;
```

Particle: Fixed. Mainnet addresses also differ from testnet.

3.2.7 Premature liquidation in `liquidateposition()` due to improper `isliquidatable` validation

Submitted by Sujith Somraaj

Severity: Medium Risk

Context: ParticlePositionManager.sol#L358

Description: The function `liquidatePosition()` in `ParticlePositionManager.sol` is used to liquidate an open position under two conditions: firstly, if the premium is not enough and secondly, if the cutoff time exceeds the loan term.

```
function isLiquidatable() {
    // ...
    return
    (tokenFromPremium < tokenFromOwed || tokenToPremium < tokenToOwed) ||
    (cutoffTime > 0 && cutoffTime + LOAN_TERM < block.timestamp);
}
```

However, during the liquidation process, the `tokenFromPremium` includes the liquidation fee, resulting in premature liquidation even if the `tokenFromPremium` is equal to or greater than the `tokenFromOwed`.

```
function  liquidatePosition(
    // ...
    closeCache.tokenFromPremium -= liquidateCache.liquidationRewardFrom;
    closeCache.tokenToPremium -= liquidateCache.liquidationRewardTo;

    if (!isLiquidatable(
        closeCache.tokenFromPremium,
        closeCache.tokenToPremium,
        liquidateCache.tokenFromOwed,
        liquidateCache.tokenToOwed,
        lien.tokenId
    )) {
    revert Errors.LiquidationNotMet();
}
```

An example scenario would be as follows: if the user has a premium of 100 USDC and the liquidation reward is 2 USDC, whereas the token owed is 100 USDC, under which the liquidation shouldn't be allowed since the user has enough premium, but due to the wrong check the user will get liquidated.

Recommendation: Move the `isLiquidatable` check before the adjustment of liquidationReward and then deduct the reward if the premium is not enough.

```
  function liquidatePosition(
      // ...
-     closeCache.tokenFromPremium -= liquidateCache.liquidationRewardFrom;
-     closeCache.tokenToPremium -= liquidateCache.liquidationRewardTo;
      if (!isLiquidatable(
           closeCache.tokenFromPremium,
           closeCache.tokenToPremium,
            liquidateCache.tokenFromOwed,
            liquidateCache.tokenToOwed,
            lien.tokenId
      )) {
          revert Errors.LiquidationNotMet();
      }
+     closeCache.tokenFromPremium -= liquidateCache.liquidationRewardFrom;
+     closeCache.tokenToPremium -= liquidateCache.liquidationRewardTo;
      // ...
  }
```

Particle: Fixed. Corrected the logic.