Polynomial Power Perp Contracts Audit

Smart Contract Security Assessment

May 11, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the Power Perp contracts component of the Polynomial protocol, an implementation of Power Perpetuals. The audited system is designed for Square Perps, i.e., perpetual-like derivatives that are designed to track the square price of an asset; e.g., ETH^2, BTC^2.

In Polynomial Power Perps, the traders can take both long and short positions against a delta-neutral AMM. Long positions are represented using an ERC-20 token and short positions are represented using ERC-721 NFTs. The current version of the AMM is built on-top of Synthetix Perps v2 to achieve delta-neutrality. Hence oracle prices are derived from Synthetix itself (via both Pyth and Chainlink oracles). Passive users can provide liquidity to the Liquidity Pool to earn exchange fees. There is an in-built funding mechanism to give fair pricing for long and short traders. This funding mechanism is baked into the pricing mechanism rather than collecting funding payments from traders.

The system also introduces the 'Kangaroo Vault' which works similarly to the liquidity pool. The vault makes a delta-neutral position by making a short Power Perp position and longing an equal amount of delta worth of Perpetuals of the underlying asset. The vault aims to profit from the funding rate and gamma of the Power Perpetuals when the underlying price diverges from the initial price.

The audit did not uncover any critical security issue that could lead to theft of users' funds. However, several high and medium severity issues related to accounting logic, financial threats such as price manipulation and bank run situations, and inefficient use of deposited funds were identified.

SETTING & CAVEATS

This audit report refers to the at the time private repository [power-perp-contracts] at commit ccbc2daba7ddec98679d8896a7022d927e2f05d6. The audited contracts were:

```
src/
  ├─ Exchange.sol
  ├─ KangarooVault.sol
  ├─ LiquidityPool.sol
  ├─ LiquidityToken.sol
  ├─ PowerPerp.sol
  ├─ ShortCollateral.sol
  ├─ ShortToken.sol
  ├─ SynthetixAdapter.sol
  ├─ SystemManager.sol
  ├─ VaultToken.sol
  ├─ interfaces/*
  ├─ libraries/
  │   └─ SignedMath.sol
  └─ utils/
      └─ PauseModifier.sol
```

Two auditors worked on the codebase for 15 days. The accompanying test suite was consulted but was not audited. The branch coverage cannot be considered sufficient for the most complex contracts as it does not exceed the 60% mark while for the `KangarooVault` it stands at 42%.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[NO CRITICAL SEVERITY ISSUES]

HIGH SEVERITY:

ID	Description	STATUS
H1	The validity of the index price, the funding rate and the mark price is not always checked by the caller	**OPEN**
Functions `getIndexPrice`, `getFundingRate` and `getMarkPrice` of the `Exchange` contract depend on the externally provided base asset price that could be invalid in some cases. Nevertheless, the aforementioned functions do not revert in case the base asset price is invalid but return a tuple with the derived value and a boolean that denotes the derived value is invalid due to the base asset price being invalid. The callers of the functions are responsible for checking the validity of the returned values, a design which is valid and flexible as long as it is appropriately implemented. However, the function `Exchange::_updateFundingRate` does not check the validity of the funding rate value returned by `getFundingRate`, which could lead to an invalid funding rate getting registered, messing up the protocol's operation. At the same time `ShortCollateral::liquidate` does not check the validity of the mark price returned by `Exchange`'s `getMarkPrice`. The chance that something will go wrong is significantly smaller with `liquidate` because each call to it is preceded by a call to the function `maxLiquidatableDebt` that checks the validity of the mark price.		
H2	`LiquidityPool` LP token price might be incorrect	**OPEN**
`LiquidityPool::getTokenPrice`, the function that computes the price of one LP token, might return an incorrect price under certain circumstances. Specifically, it is incorrectly assumed that if the `skew` is equal to `0` the `totalMargin` and `usedFunds` will always add up to `0`.		

LiquidityPool::getTokenPrice()

```
function getTokenPrice() public view override returns (uint256) {
    if (totalFunds == 0) {
        return 1e18;
    }

    uint256 totalSupply =
        liquidityToken.totalSupply() + totalQueuedWithdrawals;
    int256 skew = _getSkew();

    if (skew == 0) {
        // Dedaub: Incorrect assumption that if skew == 0 then
        //         totalMargin + usedFunds == 0
        return totalFunds.divWadDown(totalSupply);
    }

    (uint256 markPrice, bool isInvalid) = getMarkPrice();
    require(!isInvalid);

    uint256 totalValue = totalFunds;
    uint256 amountOwed = markPrice.mulWadDown(powerPerp.totalSupply());
    uint256 amountToCollect =
        markPrice.mulWadDown(shortToken.totalShorts());
    uint256 totalMargin = _getTotalMargin();

    totalValue += totalMargin + amountToCollect;
    totalValue -= uint256((int256(amountOwed) + usedFunds));

    return totalValue.divWadDown(totalSupply);
}
```

H3	The accounting of `LiquidityPool`'s queued orders is incorrect	**OPEN**

`LiquidityPool::_placeDelayedOrder` does not set the `queuedPerpSize` storage variable to `0` when an order of size `sizeDelta + queuedPerpSize` is submitted to the Synthetix Perpetual Market. Also, `queuedPerpSize` should also be accounted for in the `SubmitDelayedOrder` emitted event.

LiquidityPool::_placeDelayedOrder()

```
function _placeDelayedOrder(
    int256 sizeDelta,
    bool isLiquidation
) internal {
    PerpsV2MarketBaseTypes.DelayedOrder memory order =
      perpMarket.delayedOrders(address(this));

    (,,,,, IPerpsV2MarketBaseTypes.Status status) =
        perpMarket.postTradeDetails(sizeDelta, 0,
            IPerpsV2MarketBaseTypes.OrderType.Delayed, address(this));

    int256 oldSize = order.sizeDelta;
    if (oldSize != 0 || isLiquidation || uint8(status) != 0) {
        queuedPerpSize += sizeDelta;
        return;
    }
    perpMarket.submitOffchainDelayedOrderWithTracking(
       sizeDelta + queuedPerpSize,
       perpPriceImpactDelta,
       synthetixTrackingCode
    );
    // Dedaub: queuedPerpSize should be set to 0

    // Dedaub: Below line should be:
    //         emit SubmitDelayedOrder(sizeDelta + queuedPerpSize);
    emit SubmitDelayedOrder(sizeDelta);
}
```

H4	The mark price is susceptible to manipulation	OPEN

The mark price depends on the total sizes of the long and short positions. `ShortCollateraltoken::canLiquidate` and `maxLiquidatableDebt` use the mark price to compute the value of the position and to check if the collateralization ratio is above the liquidation limit or not. An adversary could open a large short position to increase the mark price and therefore decrease the collateral ratio of all the positions and possibly make some of them undercollateralized. The adversary would then proceed by calling `Exchange's liquidate` function to liquidate the underwater position(s) and get the liquidation bonus before finally closing their short position.

H5	Accounting of `usedFunds` and `totalFunds` is incorrect	OPEN

The `LiquidityPool` contract uses two storage variables to track its available balance, `usedFunds` and `totalFunds`. As one would expect, these two variables get updated when a position is open or closed, i.e., when functions `openLong`, `openShort`, `closeLong` and `closeShort` are called. Incoming (`openLong` and `closeShort`) and outgoing (`openShort` and `closeLong`) funds for the position must be considered together with funds needed for fees. There are 3 types of fees, trading fees attributed to the `LiquidityPool`, fees required to open an offsetting position in the Synthetix Perp Market, which are called `hedgingFees`, and a protocol fee, `externalFee`. The accounting of all these values is rather complex and ends up being incorrect in all the four aforementioned functions.

Let's take the `closeLong` function as an example. In `closeLong` there are no incoming funds and the outgoing funds are the sum of the `totalCost`, the `externalFee` and the `hedgingFees`. However, the `usedFunds` are actually increased by `tradeCost` or `totalCost+tradingFee+externalFee+hedgingFees`, while `hedgingFees` are also added to `usedFunds` in the `_hedge` function. Thus, there are two issues: (1) `hedgingFees` are accounted for twice and (2) `tradingFee` is added when it should not.

LiquidityPool::closeLong()

```
function closeLong(uint256 amount, address user, bytes32 referralCode)
    external
    override
    onlyExchange
    nonReentrant
    returns (uint256 totalCost)
{
    (uint256 markPrice, bool isInvalid) = getMarkPrice();
    require(!isInvalid);

    uint256 tradeCost = amount.mulWadDown(markPrice);
    uint256 fees = orderFee(-int256(amount));
    totalCost = tradeCost - fees;

    SUSD.safeTransfer(user, totalCost);

    uint256 hedgingFees = _hedge(-int256(amount), false);
    uint256 feesCollected = fees - hedgingFees;
    uint256 externalFee = feesCollected.mulWadDown(devFee);

    SUSD.safeTransfer(feeReceipient, externalFee);

    // Dedaub: usedFunds is incremented by tradeCost
    //         tradeCost = totalCost + fees
    //         fees = feesCollected + hedgingFees
    //         and feesCollected = tradingFee + externalFee
    usedFunds += int256(tradeCost);
    emit RegisterTrade(referralCode, feesCollected, externalFee);
    emit CloseLong(markPrice, amount, fees);
}
```

The functions `openLong`, `openShort` and `closeShort` suffer from similar issues.

H6	There might not be enough incentives for liquidators to liquidate unhealthy positions	**OPEN**

Collateralized short positions opened via the `Exchange` can get liquidated. For a liquidatable position of size `N` the liquidator has to give up `N` PowerPerp tokens for an amount of short collateral tokens equaling the value of the position plus a liquidation bonus. Thus, a user/liquidator is incentivized to liquidate a losing position instead of just closing their position, as they will get a liquidation bonus on top of what they would get. However, the liquidator might not always get paid "*an amount of short collateral tokens equaling the value of the position plus a liquidation bonus*" according to the following condition in function `ShortCollateral::liquidate`:

`ShortCollateral::liquidate()`

```
totalCollateralReturned = liqBonus + collateralClaim;
if (totalCollateralReturned > userCollateral.amount)
    totalCollateralReturned = userCollateral.amount;
```

As can be seen, if the value of the position plus the liquidation bonus, or `totalCollateralReturned`, is greater than the position's collateral, the liquidator gets just the position's collateral. This means that if during a significant price increase liquidations do not happen fast enough, certain losing positions will not be liquidatable for a profit, as the collateral's value will be less than that of the long position that needs to be closed. However, such a market is not healthy and this is reflected in the mark price, which lies in the center of the protocol. To avoid such scenarios (1) the collateralization ratios need to be chosen carefully while taking into account the squared nature of the perps and (2) an emergency fund should be implemented, which will be able to chip in when a position's collateral is not enough to incentivize its liquidation.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Liquidators are not able to set a minimum amount of collateral that they expect from a liquidation	**OPEN**

Function `Exchange::_liquidate` **does not require** that `totalCollateralReturned`, i.e., the collateral awarded to the liquidator, is greater than a liquidator-specified minimum, thus in certain cases the liquidator might get back less than what they expected (as mentioned in issue H6). This might happen because the collateral of the position is not enough to cover the liquidation's theoretical collateral claim (bad debt scenario) plus the liquidation bonus. As can be seen in the below snippet of the `ShortCollateral`'s `liquidate` function, the `totalCollateralReturned` will be at most equal to the collateral of the specific position.

`ShortCollateral::liquidate()`

```
function liquidate(uint256 positionId, uint256 debt, address user)
    external
    override
    onlyExchange
    nonReentrant
    returns (uint256 totalCollateralReturned)
{
    // Dedaub: Code omitted for brevity
    uint256 collateralClaim = debt.mulDivDown(markPrice, collateralPrice);
    uint256 liqBonus = collateralClaim.mulWadDown(coll.liqBonus);
    totalCollateralReturned = liqBonus + collateralClaim;

    // Dedaub: This if statement can reduce totalCollateralReturned to
    //         something smaller than expected by the liquidator
    if (totalCollateralReturned > userCollateral.amount)
        totalCollateralReturned = userCollateral.amount;
```

```
    userCollateral.amount -= totalCollateralReturned;
    // Dedaub: Code omitted for brevity
}
```

M2	KangarooVault funds are not optimally managed	OPEN

Function KangarooVault::_clearPendingOpenOrders determines if the previous open order has been successfully executed or has been canceled. In case the order has been canceled, the opposite exchange order is closed and the KangarooVault position data are adjusted to how they were before opening the order. However, the margin transferred to the Synthetix Perpetual Market, which was required for the position, is not revoked, meaning that the KangarooVault funds are not optimally managed. At the same time, when a pending close order's execution is confirmed in the function _clearPendingCloseOrders, the margin deposited to the Synthetix Perpetual Market is not reduced accordingly except when positionData.shortAmount == 0.

The KangarooVault funds could also be suboptimally managed because the function KangarooVault::_openPosition does not take into account the already available margin when calculating the margin needed for a new open order. If the already opened position has available margin the KangarooVault could use part of that for its new order and transfer less than what would be needed if there was no margin available.

M3	LiquidityPool and KangarooVault could be susceptible to bank runs	OPEN

The LiquidityPool and KangarooVault contracts could be susceptible to bank runs. As these two contracts can use up to their whole available balance, liquidity providers might rush to withdraw their deposits when they feel that they might not be able to withdraw for some time. At the same time, depositors would rush to withdraw if they

realized that the pool's Synthetix position is in danger and their funds that have been deposited as margin could get lost.

A buffer of funds that are always available for withdrawal could increase the trust of liquidity providers to the system. Also, an emergency fund, which is built from fees and could help alleviate fund losses, could also help make the system more robust against bank run scenarios.

M4	Casual users might be more vulnerable in a bank run	**OPEN**

In a bank run situation casual users of the `LiquidityPool` (i.e., users that interact with it through the web UI) might not be able to withdraw their funds. This is because the `LiquidityPool` offers different withdrawal functionality for different users. Power users (or protocols that integrate with the `LiquidityPool`) are expected to use the `withdraw` function, which offers immediate withdrawals for a small fee, while casual users that use the web UI will use the `queueWithdraw` function, which queues the withdrawal so it can be processed at a later time.

M5	Unsafe ERC20 transfer in `LiquidityPool::withdraw`	**OPEN**

Function `LiquidityPool::withdraw` uses a plain ERC20 transfer without checking the returned value, which is an unsafe practice. It is recommended to always either use OpenZeppelin's `SafeERC20` library or at least to wrap each operation in a `require` statement.

M6	Wrong withdrawal calculations in `KangarooVault`	**OPEN**

Function `processWithdrawalQueue` processes the `KangarooVault`'s queued withdrawals. It first checks if the available funds are sufficient to cover the withdrawal. If not, a partial withdrawal is made and the records are updated to reflect that. The `QueuedWithdraw.returnedAmount` field holds the value that has been returned to the

user thus far. However, it doesn't correctly account for partial withdrawals as the partial amount is being assigned to instead of being added to the variable.

KangarooVault::processWithdrawalQueue()

```
function processWithdrawalQueue(
    uint256 idCount
) external nonReentrant {
    for (uint256 i = 0; i < idCount; i++) {
        // Dedaub: Code omitted for brevity

        // Partial withdrawals if not enough available funds in the vault
        // Queue head is not increased
        if (susdToReturn > availableFunds) {
            // Dedaub: The withdrawn amounts should be accumulated in
            //         returnedAmount instead of being directly assigned
            current.returnedAmount = availableFunds;
            ...
        } else {
            // Dedaub: Although this branch is for full withdrawals, there
            //         may have been partial withdrawals before, so the
            //         accounting should also be cumulative here
            current.returnedAmount = susdToReturn;
            ...
        }
        queuedWithdrawalHead++;
    }
}
```

M7	LiquidityPool's exposure calculation may be inaccurate	**OPEN**

The Synthetix Perpetual Market has a two-step process for increasing/decreasing positions in which a request is submitted and remains in a pending state until it is executed by a keeper.

`LiquidityPool::_getExposure` does not consider the queued Synthetix Perp position tracked by the `queuedPerpSize` storage variable meaning that `LiquidityPool::getExposure` will return an inaccurate value when called between the submission and the execution of an order.

`LiquidityPool::_getExposure()`

```
function _getExposure() internal view returns (int256 exposure) {
    // Dedaub: queuedPerpSize should be considered in currentPosition
    int256 currentPosition = _getTotalPerpPosition();

    exposure = _calculateExposure(currentPosition);
}
```

`LiquidityPool::rebalanceMargin` does not consider `queuedPerpSize` too. The Polynomial team has mentioned that they plan to always call `placeQueuedOrder` before calling `rebalanceMargin`, thus adding a requirement that `queuedPerpSize` is equal to `0` would be enough to enforce that prerequisite.

M8	`LiquidityPool::_hedge` always adds margin to Synthetix	**OPEN**

The function `LiquidityPool::_hedge` is responsible for hedging every position opened against the `LiquidityPool` by opening the opposite position in the Synthetix Perp Market. In doing so, `_hedge` transfers an amount of funds to the Synthetix Perp Market to be used as margin for the position. However, margin does not need to be increased always, e.g., it does not need to be increased when the Synthetix Perp position is decreased because the `LiquidityPool` is hedging a long Position and thus goes short. When the absolute position size of the `LiquidityPool` in the Synthetix Perp Market is decreased, the `LiquidityPool` could remove the unnecessary margin or abstain from increasing it to account for the rare case where a Synthetix order is not executed. This together with frequent calls to the `rebalanceMargin` function would help improve the capital efficiency of the `LiquidityPool`.

LOW SEVERITY:

ID	Description	STATUS
L1	Computations that use invalid values could be avoided	**OPEN**
Functions `getIndexPrice`, `getFundingRate` and `getMarkPrice` of the `Exchange` contract depend on the externally provided base asset price that could be invalid in some cases. Even if the base asset price provided is invalid, a tuple `(value, true)` is returned where `value` is the value computed based on the invalid base asset price. However, if the base asset price is invalid, the tuple `(0, true)` could be returned while the whole computation is skipped to save gas unnecessarily spent on computing an invalid value.		
L2	A critical requirement is enforced by dependency code	**OPEN**
Function `Exchange::_openTrade`, when called with `params.isLong` set to `false` and `params.positionId` different from `0`, does not check that the `msg.sender` is the owner of the `params.positionId` short token position. This necessary requirement is later checked when `ShortToken::adjustPosition` is called. Nevertheless, we would recommend adding the appropriate require statement also as part of the function `_openTrade` as it is the one querying the position. This would also add an extra safeguard against a future code change that accidentally removes the already existing require statement.		
L3	A critical requirement is enforced by the ERC20 code	**OPEN**
In function `LiquidityPool::closeLong`, as in `openShort`, there is an outgoing flow of funds. However, there does not exist a require statement on the existence of the needed funds as in the `openShort` function. Of course, if there are not enough funds to be transferred out of the `LiquidityPool` contract the `ERC20` transfer code will cause a revert. Still, requiring that `usedFunds<=0 || totalFunds>=uint256(usedFunds)`		

makes the code more failproof. The same could be applied on function `rebalanceMargin` where there is an outgoing flow of funds towards the Synthetix Perp Market.		
L4	A critical requirement is enforced by callee code	**OPEN**
Function `ShortCollateral::collectCollateral` does not require that the provided collateral is approved (and matches the collateral of the already opened position). This could be problematic, i.e., a non-approved worthless collateral could be deposited instead, if every call to `collectCollateral` was not coupled with a call to `getMinCollateral` which enforces the aforementioned requirement. Implementing these requirements would constitute a step towards a more defensive approach, one that would make the system more bulletproof and robust even if the codebase continues to evolve and become more complicated.		
L5	Collateral approvals cannot be revoked	**OPEN**
The `ShortCollateral` contract does not implement any functionality to revoke collateral approvals, meaning that the contract owner cannot undo even an incorrect approval and would need to redeploy the contract if that were to happen. Implementing such functionality would require a lot of care to ensure no funds (collateral) are trapped in the system, i.e., cannot be withdrawn, due to the collateral approval being revoked and the withdrawal functionality being operational only for approved collaterals.		
L6	No events are emitted for several interactions	**OPEN**
• In `LiquidityPool::processWithdrawals` there is no event emitted when a withdrawal is attempted but there are `0` funds available to be withdrawn. • In `LiquidityPool::setFeeReceipient` there is no event emitted even though a relevant event is declared in the contract (`event UpdateFeeReceipient`)		

- In `LiquidityPool::executePerpOrders` there is no event emitted when the admin executes an order
- In `KangarooVault::executePerpOrders` there is no event emitted when the admin executes an order
- In `KangarooVault::receive` there is no event emitted when the contract receives ETH in contrast to the `LiquidityPool` that emits an event for this

L7	Lack of minimum deposit and withdraw amount checks allow users to spam the queues with small requests	**OPEN**

In `LiquidityPool`, users can request to deposit or withdraw any amount of tokens by calling the `queueDeposit` and `queueWithdraw` functions. Although there are checks in place to avoid registering zero-amount requests, there are no checks to ensure that someone cannot spam the queue with requests for infinitesimal amounts.

`LiquidityPool::queueDeposit()`

```
function queueDeposit(uint256 amount, address user)
    external
    override
    nonReentrant
    whenNotPaused("POOL_QUEUE_DEPOSIT")
{
    require(amount > 0, "Amount must be greater than 0");
    // Dedaub: Add a minDepositAmount check

    QueuedDeposit storage newDeposit = depositQueue[nextQueuedDepositId];
    ...
}
```

`LiquidityPool::queueWithdraw()`

```
function queueWithdraw(uint256 tokens, address user)
    external
```

```
    override
    nonReentrant
    whenNotPaused("POOL_QUEUE_WITHDRAW")
{
    require(liquidityToken.balanceOf(msg.sender) >= tokens && tokens > 0);
    // Dedaub: Add a minWithdrawAmount check
    ...
    QueuedWithdraw storage newWithdraw =
        withdrawalQueue[nextQueuedWithdrawalId];
    ...
}
```

Even though there is no clear financial incentive for someone to do this, an incentive would be to disrupt the normal flow of the protocol, and to annoy regular users, who would have to spend more gas until their requests were processed. However, the functions that process the queues can be called by anyone, including the admin, and users can also bypass the queues by directly depositing or withdrawing their tokens for a fee.

`KangarooVault` suffers from the same issue for withdrawals. For deposits, a `minDepositAmount` variable is defined and checked each time a new deposit call is made.

KangarooVault::initiateDeposit()

```
function initiateDeposit(
    address user,
    uint256 amount
) external nonReentrant {
    require(user != address(0x0));
    require(amount >= minDepositAmount);
    ...
}
```

KangarooVault::initiateWithdrawal()

```
function initiateWithdrawal(
    address user,
    uint256 tokens
) external nonReentrant {
    require(user != address(0x0));

    if (positionData.positionId == 0) {
        ...
    } else {
        require(tokens > 0, "Tokens must be greater than 0");
        // Dedaub: Add a minWithdrawAmount check here

        QueuedWithdraw storage newWithdraw =
            withdrawalQueue[nextQueuedWithdrawalId];
        ...
    }
    VAULT_TOKEN.burn(msg.sender, tokens);
}
```

L8	`LiquidityPool`'s deposit and withdraw arguments are not validated	**OPEN**

`LiquidityPool`'s `deposit` and `withdraw` functions do not require that the specified `user`, which will receive the tokens, is different from `address(0)`. The caller of the aforementioned functions might not set the parameter correctly or make the incorrect assumption that by setting it to `address(0)` it will default to `msg.sender`, leading to the tokens being sent to the wrong address. At the same time, the deposited/withdrawn amount is not required to be greater than `0`.

L9	`VaultToken::setVault` can be front-run	**OPEN**

The `VaultToken` contract declares the `setVault` function to solve the dual dependency problem between `VaultToken` and `KangarooVault`, as both require each

other's address for their initialisation. However, this function can be called by anyone, whereas the vault address can only be set once. As a result, we raise a warning here to emphasize that the `VaultToken` contract needs to be correctly initialized, as otherwise the call could be front-run or repeated (in case the initialization performed by the protocol team fails for some reason and the uninitialized variable remains unnoticed) to initialize the `vault` storage variable with a malicious Vault address.		
L10	`LiquidityPool::closeShort` should use `mulWadUp` too	**OPEN**
The `closeShort` function of the `LiquidityPool` contract has the same logic as `openLong`. `openLong` passes the rounding error cost to the user by using `mulWadUp` for the `tradeCost` calculation. However, `closeShort` does not adopt this behavior and uses `mulWadDown` for the same calculation. We recommend changing this to be the same as `openLong`.		

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	`LiquidityPool`'s and `KangarooVault`'s admin can control the leverage and margin of the position	**OPEN**

In `LiquidityPool`, the admin has increased power over its position leverage and the margin that is deposited to or withdrawn from the Synthetix Perp Market. More specifically:

First of all, the admin can arbitrarily set the leverage through the `LiquidityPool`'s `updateLeverage` function. Essentially, the risk of the `LiquidityPool` can be arbitrarily increased.

LiquidityPool::updateLeverage()

```
function updateLeverage(uint256 _leverage) external requiresAuth {
    require(_leverage >= 1e18);
    emit UpdateLeverage(futuresLeverage, _leverage);
    futuresLeverage = _leverage;
}
```

LiquidityPool::_calculateMargin()

```
function _calculateMargin(
    int256 size
) internal view returns (uint256 margin) {
    (uint256 spotPrice, bool isInvalid) = baseAssetPrice();

    require(!isInvalid && spotPrice > 0);

    uint256 absSize = size.abs();
    margin = absSize.mulDivDown(spotPrice, futuresLeverage);
}
```

The admin is also responsible for managing the margin of the pool's Synthetix Perp position. Via the `LiquidityPool::increaseMargin` function, the admin can use up to the whole available balance of the pool. The logic that decides when the aforementioned function is called is off-chain.

LiquidityPool::increaseMargin()

```
function increaseMargin(
```

```
    uint256 additionalMargin
) external requiresAuth nonReentrant {
    perpMarket.transferMargin(int256(additionalMargin));

    usedFunds += int256(additionalMargin);
    require(usedFunds <= 0 || totalFunds >= uint256(usedFunds));

    emit IncreaseMargin(additionalMargin);
}
```

Additionally, the `LiquidityPool::rebalanceMargin` function can be used to increase or decrease the pool's margin inside the limits set by the pool's leverage and the margin limits set by Synthetix. Again the logic that decides the `marginDelta` parameter and calls `rebalanceMargin` is off-chain.

The `KangarooVault` suffers from similar centralization issues. Nevertheless, the function `setLeverage` of the `KangarooVault` does not allow the admin to set the leverage to more than 5x.

N2	`LiquidityPool` admin can drain all deposited funds by being able to arbitrarily set the fee percentages	**OPEN**

In `LiquidityPool`, there are several functions that only the admin can control and allow him to parameterise all fee variables, such as deposit and withdrawal fees. However, there are no limits imposed on the values set for these variables.

`LiquidityPool::setFees()`

```
function setFees(
    uint256 _depositFee,
    uint256 _withdrawalFee
) external requiresAuth {
    ...
    // Dedaub: We recommend adding checks for depositFee and withdrawalFee
    //         to prevent unrestricted fee rates
```

```
    depositFee = _depositFee;
    withdrawalFee = _withdrawalFee;
}
```

This means that the admin could change the deposit/withdrawal fee and have all the newly deposited/withdrawn funds moved to the `feeRecipient` address. Apart from the obvious centralisation issue, such checks could prevent huge losses in the event of a compromise of the admin account or the protocol itself. On the other hand, such checks have been used in the `KangarooVault` and thus we strongly recommend adding them to `LiquidityPool` as well.

`KangarooVault::setFees()`

```
function setFees(
    uint256 _performanceFee,
    uint256 _withdrawalFee
) external requiresAuth {
    require(_performanceFee <= 1e17 && _withdrawalFee <= 1e16);
    ...
    performanceFee = _performanceFee;
    withdrawalFee = _withdrawalFee;
}
```

The same applies for the following functions that also need limits on the possible values that can be set by the admin:

- `LiquidityPool::updateLeverage()` (see also N1 for an example)
- `LiquidityPool::updateStandardSize()`
- `LiquidityPool::setBaseTradingFee()`
- `LiquidityPool::setDevFee()`
- `LiquidityPool::setMaxSlippageFee()`

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Extra requirements can be added	**INFO**
Functions `_addCollateral` and `_removeCollateral` of the `Exchange` contract do not require that `amount > 0`. Function `_liquidate` does not require that `debtRepaying > 0`.		
A2	`ShortCollateral::approveCollateral` does not check if the collateral is already approved	**INFO**
Function `ShortCollateral::approveCollateral` does not require that `collateral.isApproved == false` to disallow approving the same collateral more than once.		
A3	`LiquidityPool::hedgePositions` can return early in some cases	**INFO**
In `LiquidityPool::hedgePositions` there is no handling of the case where `newPosition` is equal to `0` and the execution can return early.		
A4	No pause logic used in `KangarooVault`	**INFO**
Core contracts of the protocol such as `LiquidityPool` and `Exchange` inherit the `PauseModifier` and use separate pause logic on several functions. In contrast, `KangarooVault`, which has an implemented logic similar to `LiquidityPool`, inherits the `PauseModifier` but it does not use the `whenNotPaused` modifier on any function.		
A5	Functions' logic can be optimized to save gas	**INFO**

In the functions `LiquidityPool::processWithdraws` and `KangarooVault::processWithdrawalQueue`, the LP token price is calculated in every iteration of the loop that processes withdrawals when in fact it does not change. Thus, the computation could be performed once, before the loop, to save gas.

A6	Unnecessary calls to `LiquidityPool` from `KangarooVault`	**INFO**

The functions `removeCollateral` and `_openPosition` of the `KangarooVault` contract, call `LiquidityPool::getMarkPrice` to get the mark price. However, this function only calls `Exchange::getMarkPrice` without adding any extra functionality. Therefore, we recommend making a direct call to `Exchange::getMarkPrice` from `KangarooVault` instead, to save some gas.

A7	Functions could be made `external`	**INFO**

The following functions could be made `external` instead of `public`, as they are not called by any of the contract functions:

Exchange.sol
- `refresh`
- `orderFee`

LiquidityPool.sol
LiquidityToken.sol
PowerPerp.sol
ShortToken.sol
- `refresh`

ShortCollateral.sol
- `refresh`
- `getMinCollateral`
- `canLiquidate`
- `maxLiquidatableDebt`

SynthetixAdapter.sol
- `getSynth`
- `getCurrencyKey`
- `getAssetPrice`
- `getAssetPrice`

SystemManager.sol
- `init`
- `setStatusFunction`

A8	Redundant overrides	INFO

All function and storage variable overrides in the `Exchange`, `LiquidityPool`, `ShortCollateral` and `SynthetixAdapter` contracts are redundant and can be removed.

A9	Unused storage variables	INFO

There is a number of storage variables that are not used:

- `Exchange:SUSD`
- `KangarooVault:maxDepositAmount`
- `LiquidityPool:addressResolver`

A10	Storage variables can be made `immutable`	INFO

The following storage variables can be made `immutable`:

`SystemManager.sol`

- `addressResolver`
- `futuresMarketManager`

`SynthetixAdapter.sol`

- `synthetix`
- `exchangeRates`

A11	`LiquidityPool::liquidate` is not used	INFO

The function `liquidate` of the `LiquidityPool` contract is not called by the `Exchange`, which is the only contract that would be able to call it. At the same time, this means that the `LiquidityPool::_hedge` function is always called with its second argument being set to `false`.

Furthermore, if this function is maintained for future use, we raise a warning here that `hedgingFees` are accounted for twice. Once by `LiquidityPool::_hedge` and another one directly inside `liquidate` function.

`LiquidityPool::liquidate()`

```
function liquidate(
```

```
    uint256 amount
) external override onlyExchange nonReentrant {
    ...
    uint256 hedgingFees = _hedge(int256(amount), true);
    // Dedaub: hedgingFees are double counted here
    usedFunds += int256(hedgingFees);
    emit Liquidate(markPrice, amount);
}
```

A12	Incorrect code comment	INFO

The code comment of `KangarooVault::saveToken` mentions "`Save ERC20 token from the vault (not SUSD or UNDERLYING)`" when there is no notion of an `UNDERLYING` token.

A13	Typo in the word *recipient*	INFO

In `LiquidityPool`, `KangarooVault` and `ILiquidityPool`, all appearances of the word *recipient* word contain a typo and are written as *rec**e**ipient*. For example, the fee recipient storage variable is written as `feeReceipient` instead of `feeRecipient`.

A14	Code duplication	INFO

The functions `canLiquidate` and `maxLiquidatableDebt` of `ShortCollateral.sol` share a large proportion of their code. For readability this part coud be included in a separate method.

A15	A large liquidation bonus percentage could lead to a decrease –instead of the expected increase- of the collateral ratio	INFO

A liquidation of a part of an underwater position is expected to increase its collateralization ratio. In a partial liquidation, the liquidator deletes part of the position

and gets collateral of the same value, but also some extra collateral as liquidation bonus. If the liquidation bonus percentage is large, the collateral ratio after the liquidation could be lower compared to the one before. The parameters of the protocol should be chosen carefully to avoid this problem. For example:

```
WIPEOUT_CUTOFF * coll.liqRatio > 1 + coll.liqBonus
```

A16	No check that `normalizationUpdate` is positive	**INFO**

The functions `getMarkPrice` and `updateFundingRate` of the `Exchange` contract compute the `normalizationUpdate` variable using the formula:

```
int256 totalFunding =
    wadMul(fundingRate, (currentTimeStamp - fundingLastUpdatedTimestamp));
int256 normalizationUpdate = 1e18 - totalFunding;
```

Although the `fundingRate` is bounded (it takes values between `-maxFunding` and `maxFunding`), the difference `currentTimeStamp - fundingLastUpdatedTimestamp` is not, therefore `totalFunding` can in principle have an arbitrarily large value, especially a value greater than `1e18` (using 18 decimals precision). The result would be a negative `normalizationUpdate` and negative mark price, which would mess all the computations of the protocol. A check that `normalizationUpdate` is positive could be added. Nevertheless, since the value of the `maxFunding` is `1e16`, the protocol has to be inactive for at least `100` days, before this issue occurs.

A17	Compiler version and possible bugs	**INFO**

The code can be compiled with Solidity 0.8.9 or higher. For deployment, we recommend no floating pragmas, but a specific version, to be confident about the baseline guarantees offered by the compiler. Version 0.8.9, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.