Opyn

Security Assessment

November 29, 2021

Prepared For:
Andrew Leone | *Opyn*
andrew@opyn.co

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Devashish Tomar | *Trail of Bits*
devashish.tomar@trailofbits.com

Jaime Iglesias | *Trail of Bits*
jaime.iglesias@trailofbits.com

Executive Summary

From November 8 to November 24, 2021, Opyn engaged Trail of Bits to review the security of the power perpetual contracts. Trail of Bits conducted this assessment over six person-weeks, with three engineers working from commit hash `427ba39` of the `squeeth-monorepo` repository.

During the first week of the assessment, we focused on understanding the codebase. We began manually reviewing the `Controller` contract and its dependencies. During the second week, we focused on reviewing the Crab strategy, its contracts, and its interactions with the other system components.

Our review resulted in eight findings ranging from high to informational severity, primarily involving missing or incorrect input validation. The most significant finding allows an attacker to create a vault that cannot be liquidated. We also found that the system does not interact safely with ERC721 tokens;, tokens could become trapped in smart contracts. Additionally, the normalization factor is unsafely computed, which can block important operations in the `Controller` contract. One high-severity issue allows users to block minting operations in the strategy immediately after it is deployed. The low-severity findings involve unprotected initialization code, the lack of access controls when users add UniswapV3 token positions, and a front-running vulnerability in withdrawal operations. Finally, the undetermined-severity finding involves the use of Solidity optimizations, which can have unintended effects on the deployed code.

In addition to the security findings, we identified code-quality issues that are not related to any particular vulnerability, which are discussed in Appendix C.

The power perpetual contracts are a work in progress with room for improvement. The codebase contains architectural complexity that reduces its readability and increases the likelihood of bugs. In particular, the strategy allows users to perform flash deposits and withdrawals, but the control flow is difficult to follow since it extensively uses callback functions in several components. These aspects of the code, in addition to the high-severity issues we identified, indicate that more bugs may be present.

We recommend that the Opyn team take the following actions before deploying the project and going live:

- Address all reported issues.
- Enhance the unit tests to minimize the use of mocks.
- Write invariants and use Echidna to test them.
- Fully document all the internal and external assumptions and invariants of the system.

- Improve the code quality by implementing the suggestions in Appendix C.
- Perform another audit to ensure that there are no further issues.

Project Dashboard

Application Summary

Name	Opyn power perpetuals
Version	`427ba39`
Type	Smart contracts
Platform	Ethereum

Engagement Summary

Dates	November 8–November 24, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■■■■
Total Medium-Severity Issues	0	
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	0	
Total Undetermined-Severity Issues	1	■
Total	8	

Category Breakdown

Data Validation	4	■■■■
Access Controls	1	■
Timing	2	■■
Undefined Behavior	1	■
Total	8	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Moderate.** The interactions between the components have sufficient access controls. However, users can send collateral to and burn the debt of any vault that they are not operators of. Even though this ability does not endanger the vaults, as these operations benefit them, it can create confusion around access controls (TOB-OPYN-PP-006).
Arithmetic	**Moderate.** The codebase uses `SafeMath` functions for calculations, and we did not identify any potential overflows in places in which these functions are not used. However, we identified one high-severity arithmetic issue (TOB-OPYN-PP-005), in which the revert of a `SafeMath` operation can block the `Controller` contract.
Assembly Use/Low-Level Calls	**Further investigation required.** A substantial amount of assembly is used throughout the math libraries, but we did not have enough time to test or verify these uses.
Decentralization	**Satisfactory.** Centralization is limited, as the owner of the `Controller` can update `feeRate` but not by more than 1%. The owner can also pause the `Controller` but for only a limited duration. The owner can shut down the `Controller` permanently in case of emergency, but users are still allowed to redeem their short and long positions.
Code Stability	**Strong.** The code was not changed during the audit.
Contract Upgradeability	**Not applicable**. The contracts are not upgradeable.
Function Composition	**Moderate.** The functions and contracts are organized and scoped appropriately. However, the workflow of the flash functions is difficult to follow, and the inline documentation that explains their workings is often not sufficient to fully understand the code.
Front-Running	**Further investigation required**. We did not have enough time to fully cover this aspect of the system; however, we found two low-severity issues related to front-running (TOB-OPYN-PP-004, TOB-OPYN-PP-008), and we suspect that more front-running issues that can affect the codebase may be present.

Monitoring	**Satisfactory.** All of the functions emit events where appropriate. The events emitted by the code are capable of effectively monitoring on-chain activity.
Specification	**Moderate.** Opyn's white paper and the inline comments throughout the codebase provide adequate documentation of the protocol. However, the inline documentation of the arithmetic computations in the `Controller` contract and the strategy is not sufficient to understand whether the code is correct.
Testing and Verification	**Moderate.** While the code has high coverage and most interactions are tested, contract mocks are used. While mocks are generally not a problem, their implementations should be as close as possible to the real contracts. However, this is not the case for some critical components, hiding a high-severity issue (TOB-OPYN-PP-002). For this reason, we recommend using mocks more restrictively and reviewing the implementations of the mocks currently in use.

Engagement Goals

We sought to answer the following questions:

- Do the system components have appropriate access controls?
- Is it possible to manipulate the system by front-running transactions?
- Is it possible for participants to steal or lose tokens or shares?
- Are there any circumstances under which arithmetic errors can affect the system?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Can flash loans negatively affect the system's operations?
- Does the arithmetic for the `Controller` and `Strategy` bookkeeping hold?
- Are critical events logged to ensure the correct operation of the system?

Coverage

The engagement focused on the following components:

- **The `Controller`:** This contract is the main entry point for users and admins of Opyn's power perpetuals product; it handles vault administration, redemptions, and liquidations. Through a manual and automated review, we assessed the soundness of the arithmetic, the accuracy of the bookkeeping operations, and the access controls that ensure that the functionality can be invoked only by the appropriate core contracts.

- **The vaults:** Vaults allow users to mint power perpetuals by pledging collateral in the form of ETH or UniswapV3 token positions they are meant to keep track of the debt users hold and to always keep collateral assets within a specific range, otherwise they become targets for liquidation. The vaults are a core component of Opyn's power perpetuals product, as they act as debt ledgers for users. Through a manual and automated review, we verified the bookkeeping operations and the access controls, and we ensured that the collateralization ratios are not broken and that liquidations work as intended.

- **The Oracle:** This is another core component of Opyn's power perpetuals product; it is used to fetch collateral and power perpetual prices so that the vaults remain within their collateralization ranges and that debt is minted using the right amount of collateral. It is also used in the Crab strategy, which is itself a vault, to perform hedging and rebalancing operations. We used a manual and automated review to verify the soundness of the arithmetic and the oracle's resistance to manipulation and censorship.

- **The Crab strategy:** The strategy relies heavily on the `Controller` contract. Because the strategy itself is a vault, it has to keep a ledger of debt and collateral. Furthermore, the strategy mints tokens to users who deposit collateral into it; the minted tokens represent a user's claim to a share of the collateral held by the strategy. The strategy also enables flash deposit, withdrawal, and hedging capabilities to facilitate arbitrage operations and to rebalance the strategy so that it can provide its users with yield. We used a manual and automated review to verify the soundness of the arithmetic and the strategy's resistance to economic manipulation.

- **Access controls:** Some parts of the system expose privileged functionalities that should be invoked only by other system contracts. We reviewed these functionalities to ensure that they can be triggered only by the intended components and that they do not contain unnecessary privileges that could be abused.

- **Arithmetic:** We reviewed the calculations for logical inconsistencies, rounding issues, and scenarios in which reverts caused by overflows could disrupt the protocol.

The off-chain components (e.g., front end, subgraph, etc.) were out of scope for this audit.

Automated Testing and Verification

To enhance coverage of certain areas of the contracts, we used automated testing techniques, including the following:

- Slither, a Solidity static analysis framework, statically verifies algebraic relationships between Solidity variables. We used Slither to detect common flaws across the codebase.
- Echidna, a smart contract fuzzer, rapidly tests security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties of certain parts of the code.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations:

- Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode.
- Echidna may not randomly generate an edge case that violates a property.

To mitigate these risks, we generate 50,000 test cases per property with Echidna and then manually review all results.

System Properties

The system properties that we tested cover properties for the `Controller` contract, vaults, and oracles to ensure that they work as expected and will not revert.

#	Property	Result
1	If the preconditions are met, the computation of the time-weighted average price does not overflow or revert.	PASSED
2	If the preconditions are met, the computation of UniswapV3 token position balances does not overflow or revert.	PASSED
3	If the preconditions are met, the computation of the normalization factor does not revert.	****FAILED** (TOB-OPYN-PP-005)**

Recommendations Summary

Short Term

❑ **Ensure that the ERC721 implementations execute the standard callback when they are required.** This will avoid trapping tokens in 3rd-party contracts interacting with the power perpetuals. (TOB-OPYN-PP-001)

❑ **Ensure that the liquidity values of UniswapV3 token positions are properly checked.** Checking only if the UniswapV3 token position has any liquidity might not be enough as it can have zero liquidity but still hold underlying assets that can be collected. **(**TOB-OPYN-PP-002)

❑ **Measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.** This will mitigate the possibility of deploying incorrect code. (TOB-OPYN-PP-003)

❑ **Implement an access control check for the `init` function of the ShortPowerPerp and WPowerPerp contracts.** These actions will help remediate the front-running of the `init` functions. (TOB-OPYN-PP-004)

❑ **Ensure the computation of the normalization factor cannot overflow and that the results are always correct.** Verify that these calculations cannot overflow and the results are always correct. (TOB-OPYN-PP-005**)**

❑ **Refactor the code in the strategy to ensure that the first call to deposit works as expected and cannot be blocked.** It currently assumes that all deposits will happen through the strategy contract and this is not correct. (TOB-OPYN-PP-006)

❑ **Add access controls to allow only the owner of a vault to deposit a UniswapV3 token position as collateral. Be mindful of the idiosyncrasies regarding access controls of vaults.** This will help in avoiding the edge case mentioned in (TOB-OPYN-PP-007)

❑ **Refactor the code so that, given a number of power perpetuals and strategy tokens, it calculates the maximum amount of collateral that can be withdrawn for the current debt without breaking the invariant.** The current implementation will revert in case the debt decreases from the time in which the transaction is sent to when the transaction is mined. This will allow transactions to succeed without breaking the invariant. (TOB-OPYN-PP-008)

Long Term

❑ **Review all ERC standards to ensure that the contracts implement them correctly.** This will avoid unexpected behavior when a third-party contract interacts with the power perpetuals. (TOB-OPYN-PP-001)

❑ **Review all the assumptions from all the DeFi applications that are integrated into the contracts.** This will mitigate the composability risk when using DeFi applications. (TOB-OPYN-PP-002)

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity**. This will help to decide if optimizations are mature enough for deployed code. (TOB-OPYN-PP-003)

❑ **Carefully review the use of initializers to make sure they have proper access controls**. This will help to catch any issue that allows unauthorized users to access them. (TOB-OPYN-PP-004)

❑ **Carefully review the effects of reverts when using SafeMath**. This will mitigate the effect of unexpected reverts that can block the contracts. (TOB-OPYN-PP-005).

❑ **Review the access controls of every operation to ensure that they cannot be exploited.** This will help to catch any issue that allows unauthorized users to access privileged operations. (TOB-OPYN-PP-007)

❑ **Carefully review the important system invariants and use Echidna to test them.** This will help to catch issues during the development process (TOB-OPYN-PP-005, TOB-OPYN-PP-006, TOB-OPYN-PP-008)

Findings Summary

#	Title	Type	Severity
1	onERC721Received callback is never called when new tokens are minted or transfered	Data Validation	High
2	Users can create vaults that cannot be liquidated	Data Validation	High
3	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
4	Initialization function can be front-run	Timing	Low
5	The computation of the normalization factor can fail	Data Validation	High
6	Users can disrupt the bookkeeping of the strategy when it is deployed	Data Validation	High
7	Lack of access controls allows anyone to deposit Uniswap tokens	Access Controls	Low
8	Front-running a withdrawal operation can cause it to revert	Timing	Low

1. onERC721Received callback is never called when new tokens are minted or transferred

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-OPYN-PP-001
Target: `Controller.sol`

Description
The ERC721 implementation used by the `Controller` contract does not properly call the corresponding callback when new tokens are minted or transferred.

The ERC721 standard states that the `onERC721Received` callback *must* be called when a mint or transfer operation occurs. However, the smart contracts interacting as users of the `ShortPowerPerp` or the `NonfungiblePositionManager` contracts (from Uniswap) will *not* be notified with the `onERC721Received` callback, as expected according to the ERC721 standard.

```
   /**
     * @notice mint new NFT
     * @dev autoincrement tokenId starts at 1
     * @param _recipient recipient address for NFT
     */
    function mintNFT(address _recipient) external onlyController returns (uint256 tokenId) {
        // mint NFT
        _mint(_recipient, (tokenId = nextId++));
    }
```

Figure 1.1: The `mintNFT` function in `ShortPowerPerp`

```
    /**
     * @notice remove uniswap v3 position token from the vault
     * @dev this function will update the vault memory in-place
     * @param _vault the Vault memory to update
     * @param _account where to send the uni position token to
     * @param _vaultId id of the vault
     */
    function _withdrawUniPositionToken(
        VaultLib.Vault memory _vault,
        address _account,
        uint256 _vaultId
    ) internal {
        _checkCanModifyVault(_vaultId, _account);

        uint256 tokenId = _vault.NftCollateralId;
        _vault.removeUniNftCollateral();
        INonfungiblePositionManager(uniswapPositionManager).transferFrom(address(this),
_account, tokenId);
        emit WithdrawUniPositionToken(msg.sender, _vaultId, tokenId);
    }
```

Figure 1.2: The `_withdrawUniPositionToken` function in `Controller`

Exploit Scenario
Alice deploys a contract to interact with the `Controller` contract to send and receive ERC721 tokens. Her contract correctly implements the `onERC71Received` callback, but this is not called when tokens are minted or transferred back to her contract. As a result, the tokens are trapped.

Recommendations
Short term, ensure that the ERC721 implementations execute the standard callback when they are required.

Long term, review all ERC standards to ensure that the contracts implement them correctly.

2. Users can create vaults that cannot be liquidated

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-OPYN-PP-002
Target: `Controller.sol`

Description
The `Controller` contract does not check that the UniswapV3 token positions deposited into the vaults hold liquidity, allowing users to create non-liquidable vaults.

The `Controller` contract allows any user to deposit UniswapV3 token positions into vaults:

```
/**
 * @notice deposit uniswap position token into a vault to increase collateral ratio
 * @param _vaultId id of the vault
 * @param _uniTokenId uniswap position token id
 */
function depositUniPositionToken(
 uint256 _vaultId,
 uint256 _uniTokenId
) external notPaused nonReentrant {
    _checkVaultId(_vaultId);
    _applyFunding();
    VaultLib.Vault memory cachedVault = vaults[_vaultId];

    _depositUniPositionToken(cachedVault, msg.sender, _vaultId, _uniTokenId);
    _writeVault(_vaultId, cachedVault);
}
```

Figure 2.1: The `depositUniPositionToken` function in `Controller`

The UniswapV3 token positions will be verified to ensure that they correspond to valid positions (power perpetuals/WETH):

```
    /**
     * @notice deposit uniswap v3 position token into a vault
     * @dev this function will update the vault memory in-place
     * @param _vault the Vault memory to update
     * @param _account account to transfer the uniswap v3 position from
     * @param _vaultId id of the vault
     * @param _uniTokenId uniswap position token id
     */
    function _depositUniPositionToken(
        VaultLib.Vault memory _vault,
        address _account,
        uint256 _vaultId,
        uint256 _uniTokenId
    ) internal {
        //get tokens for uniswap NFT
        (, , address token0, address token1, , , , , , , ) =
INonfungiblePositionManager(uniswapPositionManager).positions(_uniTokenId);
        // only check token0 and token1, ignore fee
        // if there are multiple wPowerPerp/weth pools with different fee rate, accept
// position tokens from any of them
        require(
          (token0 == wPowerPerp && token1 == weth) ||
          (token1 == wPowerPerp && token0 == weth), "C23"
        );

        _vault.addUniNftCollateral(_uniTokenId);
        INonfungiblePositionManager(uniswapPositionManager).transferFrom(_account,
address(this), _uniTokenId);
        emit DepositUniPositionToken(msg.sender, _vaultId, _uniTokenId);
    }
```

Figure 2.2: The `_depositUniPositionToken` function in `Controller`

After some time, the vaults can become “unsafe,” allowing any other user to liquidate them. When trying to liquidate an underwater vault, the `Controller` contract will first try to reduce the vault’s debt by redeeming any deposited UniswapV3 token positions. To achieve this, Uniswap’s `NonFungiblePositionManager` contract, specifically the `decreaseLiquidity` function, will be called to drain the position of all its underlying tokens.

```
    /**
     * @notice if a vault is under the 150% collateral ratio, anyone can liquidate the vault
by burning wPowerPerp
     * @dev liquidator can get back (wPowerPerp burned) * (index price) *
(normalizationFactor)  * 110% in collateral
     * @dev normally can only liquidate 50% of a vault's debt
     * @dev if a vault is under dust limit after a liquidation can fully liquidate
     * @dev will attempt to reduceDebt first, and can earn a bounty if sucessful
     * @param _vaultId vault to liquidate
     * @param _maxDebtAmount max amount of wPowerPerpetual to repay
     * @return amount of wPowerPerp repaid
     */
    function liquidate(
     uint256 _vaultId,
     uint256 _maxDebtAmount
    ) external notPaused nonReentrant returns (uint256) {
        _checkVaultId(_vaultId);
        uint256 cachedNormFactor = _applyFunding();
        VaultLib.Vault memory cachedVault = vaults[_vaultId];

        require(!_isVaultSafe(cachedVault, cachedNormFactor), "C12");

        // try to save target vault before liquidation by reducing debt
        uint256 bounty = _reduceDebt(
            cachedVault,
            IShortPowerPerp(shortPowerPerp).ownerOf(_vaultId),
            _vaultId,
            cachedNormFactor,
            true
        );

        ...
    }
```

Figure 2.3: The `Liquidate` function in `Controller`

A requirement for `decreaseLiquidity` to work is that the position's liquidity has to be above zero. However, the lack of such a check in the `Controller` contract makes it possible to call the function with a position that holds no liquidity, which results in a revert of the liquidation logic, thus preventing the vault from being liquidated.

```
    /**
     * @notice redeem uniswap v3 position in a vault for its constituent eth and wSqueeth
     * @notice this will increase vault collateral by the amount of eth, and decrease debt
by the amount of wSqueeth
     * @dev will be executed before liquidation if there's an NFT in the vault
     * @dev pays a 2% bounty to the liquidator if called by liquidate()
     * @dev will update the vault memory in-place
     * @param _vault the Vault memory to update
     * @param _owner account to send any excess
     * @param _payBounty true if paying caller 2% bounty
     * @return bounty amount of bounty paid for liquidator
     */
    function _reduceDebt(
        VaultLib.Vault memory _vault,
        address _owner,
        uint256 _vaultId,
        uint256 _normalizationFactor,
        bool _payBounty
    ) internal returns (uint256) {
        uint256 nftId = _vault.NftCollateralId;
        if (nftId == 0) return 0;

        (uint256 withdrawnEthAmount, uint256 withdrawnWPowerPerpAmount) =
_redeemUniToken(nftId);

        ...
    }
```

Figure 2.4: The `_reduceDebt` function in `Controller`

```
    /**
     * @dev redeem a uni position token and get back wPowerPerp and eth
     * @param _uniTokenId uniswap v3 position token id
     * @return wethAmount amount of weth withdrawn from uniswap
     * @return wPowerPerpAmount amount of wPowerPerp withdrawn from uniswap
     */
    function _redeemUniToken(uint256 _uniTokenId) internal returns (uint256, uint256) {
        INonfungiblePositionManager positionManager =
INonfungiblePositionManager(uniswapPositionManager);

        (, , uint128 liquidity, , ) =
VaultLib._getUniswapPositionInfo(uniswapPositionManager, _uniTokenId);

        // prepare parameters to withdraw liquidity from uniswap v3 position manager
        INonfungiblePositionManager.DecreaseLiquidityParams memory decreaseParams =
INonfungiblePositionManager
            .DecreaseLiquidityParams({
                tokenId: _uniTokenId,
                liquidity: liquidity,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
            });

        positionManager.decreaseLiquidity(decreaseParams);

        ...
    }
```

Figure 2.5: The `_redeemUniToken` function in `Controller`

```
    /// @inheritdoc INonfungiblePositionManager
    function decreaseLiquidity(DecreaseLiquidityParams calldata params)
        external
        payable
        override
        isAuthorizedForToken(params.tokenId)
        checkDeadline(params.deadline)
        returns (uint256 amount0, uint256 amount1)
    {
        require(params.liquidity > 0);
        ...
    }
```

Figure 2.6: The `decreaseLiquidity` function in Uniswap's `NonFungiblePositionManager`

Exploit Scenario
Eve creates a new vault and mints some power perpetuals using ETH as collateral. She then creates a UniswapV3 token position and drains it of its underlying liquidity. Eve deposits the drained position into the vault as collateral.

The market changes, and Eve's vault goes underwater. Alice, a liquidator, tries to liquidate Eve's position, but her transaction fails since Eve's UniswapV3 token position has zero liquidity.

Recommendations
Short term, ensure that the liquidity values of UniswapV3 token positions are properly checked.

Long term, review all the assumptions from all the DeFi applications that are integrated into the contracts.

3. Solidity compiler optimizations can be problematic

Severity: Undetermined
Type: Undefined Behavior
Target: `hardhat.config.ts`

Difficulty: High
Finding ID: TOB-OPYN-PP-003

Description

Opyn has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

```
const UNISWAP_SETTING = {
  version: "0.7.6",
  settings: {
    optimizer: {
      enabled: true,
      runs: 4000,
    },
  },
};
...
  solidity: {
    compilers: [
      UNISWAP_SETTING,
    ]
  },
...
```

Figure 3.1: The `hardhat.config.ts` file

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten-generated solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Opyn contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

4. Initialization function can be front-run

Severity: Low
Difficulty: High
Type: Configuration
Finding ID: TOB-OPYN-PP-004
Target: `ShortPowerPerp.sol`, `WPowerPerp.sol`

Description
The `ShortPowerPerp` and `WPowerPerp` contracts have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contract.

The `Intializable` contract allows to define an initialization function:

```
abstract contract Initializable {
    ...
    /**
     * @dev Modifier to protect an initializer function from being invoked twice.
     */
    modifier initializer() {
        require(_initializing || _isConstructor() || !_initialized, "Initializable: contract
is already initialized");
        ...
```

Figure 4.1: The `Initializable` base contract

In this case, the initialization function is implemented as follows:

```
 function init(address _controller) public initializer {
     require(_controller != address(0), "Invalid controller address");
     controller = _controller;
 }
```

Figure 4.2: The `init` function in `ShortPowerPerp` and `WPowerPerp`

However, since the `init` function is not protected, an attacker could front-run it and initialize the contract with malicious values.

Exploit Scenario
Alice deploys the `ShortPowerPerp` contract. Eve front-runs the contract initialization and sets her own address as the `controller` address. As a result, she gains the ability to mint new tokens. Alice is forced to redeploy the contract.

Recommendations
Short term, implement an access control check for the `init` function of the `ShortPowerPerp` and `WPowerPerp` contracts.

Long term, carefully review the use of initializers to ensure that they have proper access controls.

5. The computation of the normalization factor can fail

Severity: High
Type: Data Validation
Target: `Controller.sol`

Difficulty: High
Finding ID: TOB-OPYN-PP-005

Description
When computing the normalization factor, the system does not properly check for a corner case that could make the transaction revert, blocking interactions with the `Controller` contract.

Certain operations in the `Controller` contract trigger the recomputation of the normalization factor, an important quantity used to perform redemptions and liquidations and to check each vault's status:

```
    function _checkVault(VaultLib.Vault memory _vault, uint256 _normalizationFactor)
internal view {
        (bool isSafe, bool isDust) = _getVaultStatus(_vault, _normalizationFactor);
        require(isSafe, "C24");
        require(!isDust, "C22");
    }
```

Figure 5.1: The `_checkVault` function in `Controller`

This value is recomputed in the `_getNewNormalizationFactor` function:

```
    function _getNewNormalizationFactor() internal view returns (uint256) {
        uint32 period = block.timestamp.sub(lastFundingUpdateTimestamp).toUint32();
        ...
        uint256 rFunding = (ONE.mul(uint256(period))).div(FUNDING_PERIOD);

        // floor mark to be at least LOWER_MARK_RATIO of index
        uint256 lowerBound = index.mul(LOWER_MARK_RATIO).div(ONE);
        if (mark < lowerBound) {
            mark = lowerBound;
        } else {
            // cap mark to be at most UPPER_MARK_RATIO of index
            uint256 upperBound = index.mul(UPPER_MARK_RATIO).div(ONE);
            if (mark > upperBound) mark = upperBound;
        }

        // newNormFactor = (mark / ( (1+rFunding) * mark - index * rFunding )) *
```

```
oldNormaFactor
        uint256 multiplier =
mark.mul(ONE_ONE).div((ONE.add(rFunding)).mul(mark).sub(index.mul(rFunding))); // multiply
by 1e36 to keep multiplier in 18 decimals

        return cacheNormFactor.mul(multiplier).div(ONE);
    }
```

Figure 5.2: Part of the `_getNewNormalizationFactor` function

If `mark` is equal to the lower bound, the computation of the normalization factor relies on the following to hold:

`(1+rFunding) * mark > index * rFunding`

This prevents underflows and divide-by-zero errors when computing the multiplier.

However, if the lower bound is used for `mark` and `period` is `346279` or larger, which is approximately 4 days, this computation can fail.

Exploit Scenario
The normalization factor is not recomputed within approximately 4 days, and the price is low enough such that the computation of the normalization factor in the `Controller` contract fails, blocking the contract.

Recommendations
Short term, ensure the computation of the normalization factor cannot overflow and that the results are always correct.

Long term, carefully review the effects of reverts when using `SafeMath` and use Echidna to test important system invariants.

6. Users can disrupt the bookkeeping of the strategy when it is deployed

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-OPYN-PP-006
Target: `CrabStrategy.sol`, `Controller.sol`

Description

The strategy contains code that initializes bookkeeping operations on the first deposit, but any user can make a collateral deposit to disrupt these operations.

When the strategy performs its first deposit, the code contains a special branch to ensure that the deposit is correctly processed:

```
    function _deposit(
        address _depositor,
        uint256 _amount,
        bool _isFlashDeposit
    ) internal returns (uint256, uint256) {
        (uint256 strategyDebt, uint256 strategyCollateral) = _syncStrategyState();
        (uint256 wSqueethToMint, uint256 ethFee) = _calcWsqueethToMintAndFee(_amount,
strategyDebt, strategyCollateral);

        uint256 depositorCrabAmount = _calcSharesToMint(_amount.sub(ethFee),
strategyCollateral, totalSupply());

        if (strategyDebt == 0 && strategyCollateral == 0) {
            // store hedge data as strategy is delta neutral at this point
            // only execute this upon first deposit
            uint256 wSqueethEthPrice = IOracle(oracle).getTwap(ethWSqueethPool, wPowerPerp,
weth, TWAP_PERIOD, true);
            timeAtLastHedge = block.timestamp;
            priceAtLastHedge = wSqueethEthPrice;
        }
        ...
```

Figure 6.1: The header of the `_deposit` function in the Crab strategy

However, any user can add any amount of collateral for any vault using `deposit` in the `Controller` contract:

```
/**
 * @dev deposit collateral into a vault
 * @param _vaultId id of the vault
 */
function deposit(uint256 _vaultId) external payable notPaused nonReentrant {
    _checkVaultId(_vaultId);
    _applyFunding();
    VaultLib.Vault memory cachedVault = vaults[_vaultId];
    _addEthCollateral(cachedVault, _vaultId, msg.value);

    _writeVault(_vaultId, cachedVault);
}
```

Figure 6.2: The `deposit` function in `Controller`

If collateral is added to the strategy's vault, the bookkeeping code corresponding to the first deposit in the strategy will fail to execute, and all subsequent deposits will fail.

Exploit Scenario
Alice deploys the `Controller` and `CrabStrategy` contracts. Eve observes the deployment and adds a small amount of collateral to the strategy vault. All calls to the `deposit` function in the strategy fail.

Recommendations
Short term, refactor the code in the strategy to ensure that the first call to `deposit` works as expected and cannot be blocked.

Long term, carefully review important system invariants and use Echidna to test them.

7. Lack of access controls allows anyone to deposit Uniswap tokens

Severity: Low
Difficulty: High
Type: Access Controls
Finding ID: TOB-OPYN-PP-007
Target: `Controller.sol`

Description
The `Controller` contract allows anyone to add a UniswapV3 token position as collateral to any vault. If a vault that does not have a UniswapV3 token position is at risk of liquidation, the vault owner could consider adding a UniswapV3 token position to prevent the vault from being liquidated.

```
    function depositUniPositionToken(uint256 _vaultId, uint256 _uniTokenId) external
notPaused nonReentrant {
        _checkVaultId(_vaultId);
        _applyFunding();
        VaultLib.Vault memory cachedVault = vaults[_vaultId];

        _depositUniPositionToken(cachedVault, msg.sender, _vaultId, _uniTokenId);
        _writeVault(_vaultId, cachedVault);
    }
```

Figure 7.1: The `depositUniPositionToken` function in `Controller`

A malicious liquidator can front-run the vault owner's transaction and add a UniswapV3 token position of insufficient value to keep the vault underwater. As a result, the vault's owner will be forced to burn perpetuals, add more ETH as collateral, or withdraw an existing UniswapV3 token position to add another position of sufficient value.

Exploit Scenario
Alice creates a vault with solely ETH as collateral. During market movements, the vault requires more collateral to prevent liquidation. Alice sends a transaction to add a UniswapV3 token position as collateral. Eve, a malicious liquidator, front-runs Alice's transaction and adds another UniswapV3 token position of lesser value, thus preventing Alice's effort to add the UniswapV3 token position as collateral.

Recommendations
Short term, add access controls to allow only the owner of a vault to deposit a UniswapV3 token position as collateral. Be mindful of the idiosyncrasies regarding access controls of vaults.

Long term, review the access controls of every operation to ensure that they cannot be exploited.

8. Front-running a withdrawal operation can cause it to revert

Severity: Low
Type: Timing
Target: `CrabStrategy.sol`

Difficulty: High
Finding ID: TOB-OPYN-PP-008

Description

An attacker can front-run the `withdraw` function in the `CrabStrategy` contract to force the transaction to fail.

The `CrabStrategy` defines the following `withdraw` function:

```
    function withdraw(uint256 _crabAmount, uint256 _wSqueethAmount) external payable
nonReentrant {
        uint256 ethToWithdraw = _withdraw(msg.sender, _crabAmount, _wSqueethAmount, false);

        // send back ETH collateral
        payable(msg.sender).sendValue(ethToWithdraw);

        emit Withdraw(msg.sender, _crabAmount, _wSqueethAmount, ethToWithdraw);
    }
```

Figure 8.1: The `withdraw` function in the `CrabStrategy` contract

The operation calculates the amount of collateral that can be withdrawn based on the amount of shares the user wants to burn.

Furthermore, if the strategy holds any debt, the strategy requires that a proportionate amount of power perpetuals be burned to keep the collateral ratio within the threshold; this amount has to be calculated by the user before performing the withdrawal.

```
    function _withdraw(
        address _from,
        uint256 _crabAmount,
        uint256 _wSqueethAmount,
        bool _isFlashWithdraw
    ) internal returns (uint256) {
        (uint256 strategyDebt, uint256 strategyCollateral) = _syncStrategyState();

        uint256 strategyShare = _calcCrabRatio(_crabAmount, totalSupply());
        uint256 ethToWithdraw = _calcEthToWithdraw(strategyShare, strategyCollateral);
        if (strategyDebt > 0) require(_wSqueethAmount.wdiv(strategyDebt) == strategyShare,
"invalid ratio");
        ...
}
```

Figure 8.2: The `_withdraw` function in the `CrabStrategy` contract

This last invariant introduces the front-running vulnerability. The user needs to calculate beforehand the amount of power perpetuals and strategy tokens they have to burn for the strategy's current debt; however, any transaction that precedes the withdrawal could change the strategy's current debt, making the withdrawal operation fail.

Exploit Scenario
Alice is a user of the Crab strategy and wants to withdraw some of her funds. She calls `withdraw`, defining a number of strategy tokens and power perpetuals to burn with her withdrawal.

Eve sees Alice's unconfirmed transaction and front-runs it with another transaction that causes the `CrabStrategy` debt to decrease. This reduction in debt causes the `require(_wSqueethAmount.wdiv(strategyDebt) == strategyShare` invariant to fail, since Alice had already defined a number of power perpetuals to burn.

Recommendations
Short term, refactor the code so that, given a number of power perpetuals and strategy tokens, it calculates the maximum amount of collateral that can be withdrawn for the current debt without breaking the invariant.

Long term, review all system invariants and use Echidna to test them.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is

	important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing and Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General Recommendations

- **Consider removing unused imports from the `Controller`, `VaultLib`, and `StrategyFlashSwap` contracts.**
- **Use consistent language across the codebase**.
  - The terms "squeeth" and "PowerPerp" are used interchangeably across the codebase, which can lead to confusion even if they are referring to the same concept. Keeping the language consistent will make the code easier to modify, review, and maintain.

`Controller.sol`

- **Consider refactoring the `_amountIn` parameter in `_fetchRawTwap` and `_fetchHistoricTwap`.** This parameter takes only the value `ONE` and can be replaced by a constant. Removing unused parameters will make the codebase easier to maintain, modify, and audit.
- **Revise and expand incomplete NatSpec comments.**
  - In the `_getFee` function, a comment about the `depositAmountAfterFee` return value is missing.
  - In the `_openDepositMint` function, comments regarding return values seem incomplete and have spelling errors.

B. Fix Log

Opyn addressed issues TOB-OPYN-PP-001 to TOB-OPYN-PP-008 in their codebase as a result of the assessment. Each of the fixes was verified by Trail of Bits. The reviewed code is available in several PRs (590, 622, 630, 642, 643, 691).

ID	Title	Severity	Status
01	onERC721Received callback is never called when new tokens are minted or transfered	High	Fixed
02	Users can create vaults that cannot be liquidated	High	Fixed
03	Solidity compiler optimizations can be dangerous	Undetermined	Not fixed
04	Initialization function can be front-run	Low	Fixed
05	The computation of the normalization factor can fail	High	Fixed
06	Users can disrupt the bookkeeping of the strategy when it is deployed	High	Fixed
07	Lack of access controls allows anyone to deposit Uniswap tokens	Low	Fixed
08	Front-running a withdrawal operation can cause it to revert	Low	Fixed

Detailed Fix Log

This section includes brief descriptions of fixes implemented by Opyn after the end of this assessment that were reviewed by Trail of Bits.

Finding 1: onERC721Received callback is never called when new tokens are minted or transfered
This appears to be resolved correctly using `safeMint` and `safeTransferFrom` when required at each mint and transfer.

Finding 2: Users can create vaults that cannot be liquidated
This appears to be resolved by verifying that the Uniswap NFT has a positive amount of liquidity.

Finding 3: Solidity compiler optimizations can be dangerous
This issue was not fixed.

Finding 4: Initialization function can be front-run
This appears to be resolved by only allowing the deployer address to call the initialization function.

Finding 5: The computation of the normalization factor can fail
This appears to be resolved by reimplementing the normalization factor computation. While the new code can still overflow and revert, this is not expected to happen in the context of valid system state.

Finding 6: Users can disrupt the bookkeeping of the strategy when it is deployed
This appears to be fixed by adding access controls to the deposit/withdrawal functions for the vault. Since the owner of the strategy vault will be a smart contract, it will never give access to the deposit functions to any other user.

Finding 7: Lack of access controls allows anyone to deposit Uniswap tokens
This appears to be fixed by adding access controls to the deposits/withdraw UNI position token functions.

Finding 8: Front-running a withdrawal operation can cause it to revert
This appears to be fixed by removing the need for two dependent parameters in order to perform a withdrawal. This will make it impossible for a front-runner to force a revert, making another withdraw.