SMART CONTRACT AUDIT REPORT

for

DelegatedPositionVault (Hyperswap)

Prepared By: Xiaomi Huang

PeckShield

June 27, 2025

Document Properties

Client	Hyperswap
Title	Smart Contract Audit Report
Target	Hyperswap DelegatedPositionVault
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 27, 2025	Xuxian Jiang	Final Release

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the DelegatedPositionVault in Hyperswap, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About DelegatedPositionVault

DelegatedPositionVault is a smart contract that allows liquidity providers to delegate their Uniswap V2/V3 positions to the vault, which mints an ERC721 NFT representing ownership of the position. The NFT holder can then claim fees from the underlying position while the vault manages the actual liquidity. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Hyperswap DelegatedPositionVault

Item	Description
Name	Hyperswap
Type	Ethereum Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 27, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/HyperSwap-Labs/burn-delegate-fees.git (ae6c710)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/HyperSwap-Labs/burn-delegate-fees.git (a4b50f2)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `DelegatedPositionVault` contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■■
Informational	0	
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Hyperswap DelegatedPositionVault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited _claimV2() Logic in DelegatedPositionVault	Business Logic	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contract is being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited _claimV2() Logic in DelegatedPositionVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DelegatedPositionVault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DelegatedPositionVault` contract allows the user to claim fees from the locked `UniswapV2/V3` position. While reviewing the logic to claim the accrued fee from a locked `UniswapV2` position, we notice an alternative, more gas-friendly approach.

To elaborate, we show below the related `_claimV2()` routine. Given a delegated `UniswapV2` position `NFT ID`, this routine claims fees from this specific position. We notice current logic burns the entire position liquidity, claims the fee, and then adds remaining funds as new liquidity. This process can be simplified by only burning the fee-proportional liquidity, i.e., `_feeFraction(metadata.sqrtK, Math.sqrt(pair.kLast()));`. By doing so, we only need to specify the recipient to the position owner when the fee-proportional liquidity is burned.

```
165     function _claimV2(uint256 tokenId) private {
166         DelegatedPosition storage position = delegatedPositions[tokenId];
167         LPV2Metadata memory metadata = position.lpV2Metadata;
168         IHyperswapV2Pair pair = metadata.pair;
169
170         require(pair.transfer(address(pair), position.tokenIdOrAmountLpTokens), "
                Transfer failed");
171         (uint256 amount0, uint256 amount1) = pair.burn(address(this));
172
173         uint256 f = _feeFraction(metadata.sqrtK, Math.sqrt(pair.kLast()));
174         uint256 fees0 = (amount0 * f) / 1e18;
175         uint256 fees1 = (amount1 * f) / 1e18;
176
```

```
177         if (fees0 > 0) {
178             require(metadata.token0.transfer(msg.sender, fees0), "Transfer failed");
179         }
180         if (fees1 > 0) {
181             require(metadata.token1.transfer(msg.sender, fees1), "Transfer failed");
182         }
183
184         uint256 principal0 = amount0 - fees0;
185         uint256 principal1 = amount1 - fees1;
186
187         IERC20(pair.token0()).approve(address(router), principal0);
188         IERC20(pair.token1()).approve(address(router), principal1);
189         (,, uint256 liquidityMinted) = router.addLiquidity(
190             address(position.lpV2Metadata.token0),
191             address(position.lpV2Metadata.token1),
192             principal0,
193             principal1,
194             0,
195             0,
196             address(this),
197             block.timestamp
198         );
199
200         position.tokenIdOrAmountLpTokens = liquidityMinted;
201
202         position.lpV2Metadata.sqrtK = Math.sqrt(pair.kLast());
203
204         emit Claim(position.tokenIdOrAmountLpTokens, PositionType.V2, fees0, fees1);
205     }
```

Listing 3.1: `DelegatedPositionVault::_claimV2()`

Recommendation Simplify the above routine to properly claim fee for a delegated `UniswapV2` position. An example revision is shown below:

```
165     function _claimV2(uint256 tokenId) private {
166         DelegatedPosition storage position = delegatedPositions[tokenId];
167         LPV2Metadata memory metadata = position.lpV2Metadata;
168         IHyperswapV2Pair pair = metadata.pair;
169
170         uint256 sqrtKNow = Math.sqrt(pair.kLast());
171         uint256 f = _feeFraction(metadata.sqrtK, sqrtKNow);
172         uint256 feeLpTokens = position.tokenIdOrAmountLpTokens * f / 1e18;
173
174         require(pair.transfer(address(pair), feeLpTokens), "Transfer failed");
175         (uint256 fees0, uint256 fees1) = pair.burn(address(msg.sender));
176
177         position.tokenIdOrAmountLpTokens -= feeLpTokens;
178
179         position.lpV2Metadata.sqrtK = sqrtKNow;
180
181         emit Claim(position.tokenIdOrAmountLpTokens, PositionType.V2, fees0, fees1);
```

```
182     }
```

Listing 3.2: Revised `DelegatedPositionVault::_claimV2()`

Status The issue has been fixed by this commit: `a4b50f2`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `DelegatedPositionVault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
64      function transfer(address _to, uint _value) returns (bool) {
65          //Default assumes totalSupply can't be over max (2^256 - 1).
66          if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67              balances[msg.sender] -= _value;
68              balances[_to] += _value;
69              Transfer(msg.sender, _to, _value);
70              return true;
71          } else { return false; }
72      }
73
74      function transferFrom(address _from, address _to, uint _value) returns (bool) {
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
```

```
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.3: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `emergencyWithdrawERC20()` routine in the `DelegatedPositionVault` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transfer(to, amount)` (line 298) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
295     function emergencyWithdrawERC20(IERC20 token, address to, uint256 amount) external
            onlyOwner whenPaused {
296         require(to != address(0), "Cannot withdraw to zero address");
297         require(amount > 0, "Amount must be greater than zero");
298         require(token.transfer(to, amount), "Transfer failed");
299     }
```

Listing 3.4: `DelegatedPositionVault::emergencyWithdrawERC20()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `a4b50f2`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DelegatedPositionVault`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `DelegatedPositionVault` smart contract, there is a privileged account `owner` that plays a critical role in governing and regulating the contract-wide operations (e.g., pause/unpause the contract and perform emergency withdrawal). The account also has the privilege to control or govern the flow

of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
274     function pause() external onlyOwner {
275         _pause();
276     }
277
278     /**
279      * @notice Unpauses all contract operations
280      * @dev Only callable by the contract owner
281      * @dev Re-enables burnV3, burnV2, and claim functions
282      * @dev Disables emergency withdrawal functions
283      */
284     function unpause() external onlyOwner {
285         _unpause();
286     }
287
288     /**
289      * @notice Emergency withdrawal of ERC20 tokens stuck in the contract
290      * @param token The ERC20 token to withdraw
291      * @param to The recipient address
292      * @param amount The amount to withdraw
293      * @dev Only callable by owner when contract is paused
294      */
295     function emergencyWithdrawERC20(IERC20 token, address to, uint256 amount) external
         onlyOwner whenPaused {
296         require(to != address(0), "Cannot withdraw to zero address");
297         require(amount > 0, "Amount must be greater than zero");
298         require(token.transfer(to, amount), "Transfer failed");
299     }
300
301     /**
302      * @notice Emergency withdrawal of ERC721 tokens (including V3 positions) stuck in
          the contract
303      * @param token The ERC721 contract address
304      * @param to The recipient address
305      * @param tokenId The token ID to withdraw
306      * @dev Only callable by owner when contract is paused. Use with extreme caution.
307      * @dev WARNING: Withdrawing locked V3 positions breaks the permanent lock guarantee
308      */
309     function emergencyWithdrawERC721(IERC721 token, address to, uint256 tokenId)
         external onlyOwner whenPaused {
310         require(to != address(0), "Cannot withdraw to zero address");
311         token.safeTransferFrom(address(this), to, tokenId);
312     }
313
314     /**
315      * @notice Emergency withdrawal of native ETH stuck in the contract
316      * @param to The recipient address
317      * @param amount The amount to withdraw
318      * @dev Only callable by owner when contract is paused
```

```
319      */
320     function emergencyWithdrawETH(address payable to, uint256 amount) external onlyOwner
             whenPaused {
321         require(to != address(0), "Cannot withdraw to zero address");
322         require(amount > 0, "Amount must be greater than zero");
323         require(address(this).balance >= amount, "Insufficient balance");
324         (bool success,) = to.call{value: amount}("");
325         require(success, "ETH transfer failed");
326     }
```

Listing 3.5: Example Privileged Functions in `DelegatedPositionVault`

Note that if these privileged accounts are plain EOA accounts, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the DelegatedPositionVault smart contract that allows liquidity providers to delegate their Uniswap V2/V3 positions to the vault, which mints an ERC721 NFT representing ownership of the position. The NFT holder can then claim fees from the underlying position while the vault manages the actual liquidity. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.