# SMART CONTRACT AUDIT REPORT

for

# Revert Lend

Prepared By: Xiaomi Huang

PeckShield

August 16, 2024

## Document Properties

| | |
|---|---|
| Client | Revert Finance |
| Title | Smart Contract Audit Report |
| Target | Revert Lend |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jason Shen, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | August 16, 2024 | Xuxian Jiang | Final Release |
| 1.0-rc | May 25, 2024 | Xuxian Jiang | Release Candidate #1 |

## Contact

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



| | |
|---|---|
| Name | Xiaomi Huang |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Revert Lend protocol, 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 Revert Lend

Revert Lend is a decentralized lending protocol specifically designed for Automated Market Maker (AMM ) Liquidity Providers on UniswapV3. This protocol facilitates the acquisition of ERC20 token loans by leveraging their liquidity provider positions as collateral, while uniquely allowing them to retain control and management of their capital within the UniswapV3 pools. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Revert Lend

| Item | Description |
|---:|---|
| Name | Revert Finance |
| Website | https://revert.finance// |
| Type | EVM Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | August 16, 2024 |

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

- https://github.com/revert-finance/lend.git (a9925b2, 4c39d7d)

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

- https://github.com/revert-finance/lend.git (ce43467, da1b1a2)

## 1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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 classified into four categories accordingly, 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

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 |

contract is considered safe regarding the check item. For any discovered issue, we might further 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) [8], 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.

## 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 design and implementation of the Revert Lend protocol. 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 | 4 | ■ ■ ■ ■ |
| Informational | 1 | ■ |
| Total | 6 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. 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 that 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, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key Revert Lend Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Informational | Insufficient Caller Validation of UniswapV3 Flashloan Callback | Security Features | Resolved |
| PVE-002 | Low | Suggested Adherence Of Checks-Effects-Interactions Pattern | Time And State | Resolved |
| PVE-003 | Low | Improved Gas Efficiency in V3Vault::_calculateGlobalInterest() | Coding Practices | Resolved |
| PVE-004 | Low | Preservation of Transform Continuity in V3Vault::transform() | Business Logic | Resolved |
| PVE-005 | Low | Improved Caller Validation in Supported Transformers | Coding Practices | Resolved |
| PVE-006 | Medium | Trust Issue on Admin Keys | Security Features | Mitigated |

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Insufficient Caller Validation of UniswapV3 Flashloan Callback

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FlashloanLiquidator`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

### Description

The `Revert Lend` protocol has a built-in flashloan support by leveraging the liquidity from `Uniswap V3` for liquidation. In the process of examining the flashloan support, we notice the related flashloan implementation does not properly enforce the caller validation.

In the following, we show below the implementation of the related callback, i.e., `uniswapV3FlashCallback()`. We notice it does not attempt to enforce the caller is from the intended `Uniswap V3 pool`. To fix, we suggest to re-calculate the pool address from the passed `tokenId` and validate the caller is from the intended `Uniswap V3 pool`. Fortunately, this `FlashloanLiquidator` contract by design is not supposed to hold any asset.

```
    function uniswapV3FlashCallback(uint256 fee0, uint256 fee1, bytes calldata callbackData) external override {
        // no origin check is needed - because the contract doesn't hold any funds - there is no benefit in calling uniswapV3FlashCallback() from another context

        FlashCallbackData memory data = abi.decode(callbackData, (FlashCallbackData));

        SafeERC20.safeIncreaseAllowance(data.asset, address(data.vault), data.liquidationCost);
        data.vault.liquidate(
            IVault.LiquidateParams(
                data.tokenId, data.swap0.amountIn, data.swap1.amountIn, address(this), "", data.deadline
```

```
            )
        );
        SafeERC20.safeApprove(data.asset, address(data.vault), 0);
    ...
}
```

Listing 3.1: `FlashloanLiquidator::uniswapV3FlashCallback()`

**Recommendation** Improve the above `uniswapV3FlashCallback()` routine to ensure it is invoked from the intended `UniswapV3 flashloan pool`.

**Status** This issue has been resolved as the team confirms it is part of the design. Specifically, this helper contract by design does not hold any asset or position.

## 3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `V3Vault`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

### Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the `Uniswap/Lendf.Me` hack [11].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `V3Vault` as an example, the `liquidate()` function (see the code snippet below) is provided to externally call token contracts a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 783) starts before effecting the update on internal states (line 792), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be abused of launching `re-entrancy` via the same entry function.

```
        debtSharesTotal = debtSharesTotal - debtShares;

        dailyDebtIncreaseLimitLeft = dailyDebtIncreaseLimitLeft + state.debt;

        // send promised collateral tokens to liquidator
        (amount0, amount1) = _sendPositionValue(
            params.tokenId, state.liquidationValue, state.fullValue, state.feeValue,
                params.recipient, params.deadline
        );

        if (amount0 < params.amount0Min  amount1 < params.amount1Min) {
            revert SlippageError();
        }

        // remove debt from loan
        _cleanupLoan(params.tokenId, state.newDebtExchangeRateX96, state.
            newLendExchangeRateX96);
```

Listing 3.2: `V3Vault::liquidate()`

Note that another routine in the same contract shares the same issue, i.e., `_deposit()`.

**Recommendation** Revisit the above routine to follow the best practice of the `checks-effects-interactions` pattern. In the meantime, we suggest the use of `nonReentrant` to effectively block this specific risk.

**Status** This issue has been resolved by the following commit: `3dbecb2`. The team further clarifies that only legit tokens will be added for asset and collateral tokens. With that, there is no reentrancy possible. Also there will be a timelock for privileged admin operations (like adding a new collateral token).

## 3.3 Improved Gas Efficiency in V3Vault::_calculateGlobalInterest()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `V3Vault`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

As mentioned earlier, `Revert Lend` is a decentralized lending protocol specifically designed for `UniswapV3 LPs`. In the process of examining the interest accrual logic, we notice a possible improvement in current implementation.

To elaborate, we show below the implementation of the related `_calculateGlobalInterest()` routine. As the name indicates, this routine is used to calculate the global interest, especially the latest `lastDebtExchangeRateX96` and `lastLendExchangeRateX96` states. We notice an internal `if` condition (line 1284), which only yields true for the first time. In fact, the intended comparision should be made with `timeElapsed`, not `lastRateUpdate`.

```
1264     function _calculateGlobalInterest()
1265         internal
1266         view
1267         returns (uint256 newDebtExchangeRateX96, uint256 newLendExchangeRateX96)
1268     {
1269         uint256 oldDebtExchangeRateX96 = lastDebtExchangeRateX96;
1270         uint256 oldLendExchangeRateX96 = lastLendExchangeRateX96;
1271 
1272         (uint256 balance,) = _getBalanceAndReserves(oldDebtExchangeRateX96,
             oldLendExchangeRateX96);
1273 
1274         uint256 debt = _convertToAssets(debtSharesTotal, oldDebtExchangeRateX96, Math.
             Rounding.Up);
1275 
1276         (uint256 borrowRateX64, uint256 supplyRateX64) = interestRateModel.
             getRatesPerSecondX64(balance, debt);
1277 
1278         supplyRateX64 = supplyRateX64.mulDiv(Q32 - reserveFactorX32, Q32);
1279 
1280         // always growing or equal
1281         uint256 lastRateUpdate = lastExchangeRateUpdate;
1282         uint256 timeElapsed = (block.timestamp - lastRateUpdate);
1283 
1284         if (lastRateUpdate != 0) {
1285             newDebtExchangeRateX96 = oldDebtExchangeRateX96 + oldDebtExchangeRateX96 *
                 timeElapsed * borrowRateX64 / Q64;
1286             newLendExchangeRateX96 = oldLendExchangeRateX96 + oldLendExchangeRateX96 *
                 timeElapsed * supplyRateX64 / Q64;
1287         } else {
1288             newDebtExchangeRateX96 = oldDebtExchangeRateX96;
1289             newLendExchangeRateX96 = oldLendExchangeRateX96;
1290         }
1291     }
```

Listing 3.3: `V3Vault::_calculateGlobalInterest()`

**Recommendation** Revisit the above `_calculateGlobalInterest()` routine to update `lastDebtExchangeRateX96` and `lastLendExchangeRateX96` only when necessary.

**Status** This issue has been fixed by the following commit: `3dbecb2`.

## 3.4 Preservation of Transform Continuity in V3Vault::transform()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: V3Vault
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

As a decentralized lending protocol, Revert Lend specializes the support of UniswapV3 LPs as collateral while uniquely allowing their owners to retain control and management of their capital within the Uniswap v3 pools. Specifically, it supports the so-called position transformers, which are specialized, protocol-approved contracts to offer enhanced flexibility in managing collateral positions. Once authorized by the collateral position owner, these transformers can modify or even completely replace a position, provided that the final status of the Collateralized Debt Position (CDP) vault is healthy. In the process of examining the related transformer logic, we notice a possible improvement in current implementation.

To elaborate, we show below the implementation of the related transform() routine. While it properly achieves the intended position transformers feature, we notice a new position may be created and the new position may be not approved for the same transformer. In other words, if a position inside V3Vault is configured to be adjusted with AutoRange, when the new position is replacing the old position (in onERC721Received()), the new position will not be able to be processed by AutoRange anymore. The reason is that it does not re-adjust the needed permission in the transformApprovals mapping.

```
    function transform(uint256 tokenId, address transformer, bytes calldata data)
        external
        override
        returns (uint256 newTokenId)
    {
        if (tokenId == 0  !transformerAllowList[transformer]) {
            revert TransformNotAllowed();
        }
        if (transformedTokenId != 0) {
            revert Reentrancy();
        }
        transformedTokenId = tokenId;

        (uint256 newDebtExchangeRateX96,) = _updateGlobalInterest();

        address loanOwner = tokenOwner[tokenId];
```

```

        // only the owner of the loan or any approved caller can call this
        if (loanOwner != msg.sender && !transformApprovals[loanOwner][tokenId][msg.sender]) {
            revert Unauthorized();
        }

        // give access to transformer
        nonfungiblePositionManager.approve(transformer, tokenId);

        (bool success,) = transformer.call(data);
        if (!success) {
            revert TransformFailed();
        }

        // may have changed in the meantime
        tokenId = transformedTokenId;

        // check owner not changed (NEEDED because token could have been moved somewhere else in the meantime)
        address owner = nonfungiblePositionManager.ownerOf(tokenId);
        if (owner != address(this)) {
            revert Unauthorized();
        }

        // remove access for transformer
        nonfungiblePositionManager.approve(address(0), tokenId);

        uint256 debt = _convertToAssets(loans[tokenId].debtShares, newDebtExchangeRateX96, Math.Rounding.Up);
        _requireLoanIsHealthy(tokenId, debt, false);

        transformedTokenId = 0;

        return tokenId;
    }
```

Listing 3.4: `V3Vault::transform()`

**Recommendation** Revisit the above `transform()` routine to adjust the `transformApprovals` mapping when a new position is created.

**Status** This issue has been fixed by the following commit: `3dbecb2`.

## 3.5 Improved Caller Validation in Supported Transformers

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LeverageTransformer
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

### Description

The Revert Lend protocol has designed a number of transformers. While examining the support of these transformers, we notice a specific one can be improved to verify the caller is from the intended vault.

In the following, we show below the code snippet of the related leverageUp() routine from a transformer named LeverageTransformer. As the name indicates, this routine is designed to leverage positions directly in one single transaction. We notice the caller is being validated with a helper validateCaller(), which unfortunately does not ensure the caller is from a supported vault. With that, we suggest to strengthen the caller verification by explicitly checking require(vaults[msg.sender ]). Note another routine leverageDown() from the same contract shares the same issue.

```
    function leverageUp(LeverageUpParams calldata params) external {
        _validateCaller(nonfungiblePositionManager, params.tokenId);

        uint256 amount = params.borrowAmount;

        address token = IVault(msg.sender).asset();

        IVault(msg.sender).borrow(params.tokenId, amount);
        ...
  }
```

Listing 3.5: LeverageTransformer::leverageUp())

**Recommendation** Improve the caller verification in the above-mentioned routines to ensure the caller is a legitimate vault in the Revert Lend protocol.

**Status** This issue has been resolved. The team considers there is no need and our suggestion only serves as a precaution mechanism.

## 3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

### Description

The `Revert Lend` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, manage transformers, transform positions, as well as withdraw reserves). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function withdrawReserves(uint256 amount, address receiver) external onlyOwner {
      ...
    }
    ...
    function setTransformer(address transformer, bool active) external onlyOwner {
        // protects protocol from owner trying to set dangerous transformer
        if (
            transformer == address(0)  transformer == address(this)  transformer ==
                asset
                 transformer == address(nonfungiblePositionManager)
        ) {
            revert InvalidConfig();
        }

        transformerAllowList[transformer] = active;
        emit SetTransformer(transformer, active);
    }
    ...
    function setLimits(
        uint256 _minLoanSize,
        uint256 _globalLendLimit,
        uint256 _globalDebtLimit,
        uint256 _dailyLendIncreaseLimitMin,
        uint256 _dailyDebtIncreaseLimitMin
    ) external {
      ...
    }
    ...
    function setReserveFactor(uint32 _reserveFactorX32) external onlyOwner {
        // update interest to be sure that reservefactor change is applied from now on
        _updateGlobalInterest();
        reserveFactorX32 = _reserveFactorX32;
```

```
        emit SetReserveFactor(_reserveFactorX32);
    }
    ...
    function setReserveProtectionFactor(uint32 _reserveProtectionFactorX32) external
        onlyOwner {
        if (_reserveProtectionFactorX32 < MIN_RESERVE_PROTECTION_FACTOR_X32) {
            revert InvalidConfig();
        }
        reserveProtectionFactorX32 = _reserveProtectionFactorX32;
        emit SetReserveProtectionFactor(_reserveProtectionFactorX32);
    }
    ...
    function setTokenConfig(address token, uint32 collateralFactorX32, uint32
        collateralValueLimitFactorX32)
        external
        onlyOwner {...}

    /// @notice Updates emergency admin address (onlyOwner)
    /// @param admin Emergency admin address
    function setEmergencyAdmin(address admin) external onlyOwner {
        emergencyAdmin = admin;
        emit SetEmergencyAdmin(admin);
    }
```

Listing 3.6: Example Privileged Operations in `V3Vault`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

**Recommendation** Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance. And revisit the weakened trust model to ensure the `multisig` support is not reduced.

**Status** This issue has been mitigated as the ownership will be controlled through a timelock contract. Also, emergency admin role will be a multi-sig only.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the Revert Lend protocol, which is a decentralized lending protocol specifically designed for Automated Market Maker (AMM) Liquidity Providers on UniswapV3. This protocol facilitates the acquisition of ERC20 token loans by leveraging their liquidity provider positions as collateral, while uniquely allowing them to retain control and management of their capital within the UniswapV3 pools. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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

[11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[12] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.