SMART CONTRACT AUDIT REPORT

for

DeltaPrime

Prepared By: Xiaomi Huang

PeckShield

May 27, 2023

Document Properties

Client	DeltaPrimeLabs
Title	Smart Contract Audit Report
Target	DeltaPrime
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 27, 2023	Xuxian Jiang	Final Release
1.0-rc2	May 23, 2023	Xuxian Jiang	Release Candidate #2
1.0-rc	May 3, 2023	Xuxian Jiang	Release Candidate

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 DeltaPrime 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 DeltaPrime

DeltaPrime is a lending platform on Avalanche that will allow under-collateral borrowing from pooled deposits. The key mechanism is to enable fund-lending not to a personal account, but a special purpose smart-contract. The contract automatically guards solvency and every activity needs to undergo a series of checks. The insolvency risk is further mitigated by a decentralized liquidation mechanism allowing anyone to forcibly repay part of the loan due to assets price movements caused by external factors. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DeltaPrime

Item	Description
Issuer	DeltaPrimeLabs
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 27, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Smart Loans assumes a trusted price oracle with timely market price feeds for supported assets. The oracle is not part of this audit.

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans.git (e0b5c3b)

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

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans.git (147c777)

1.2 About PeckShield

PeckShield Inc. [11] 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 the OWASP Risk Rating Methodology [10]:

- 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	DeltaPrimeLabs 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

To evaluate the risk, we go through a checklist of 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 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) [9], 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 Logic	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 `DeltaPrime` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	0	
Low	5	■■■■■
Informational	0	
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 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 high-severity vulnerability and 5 low-severity vulnerabilities.

Table 2.1: Key DeltaPrime Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Sybil-Based Attack to Steal Vesting Reward	Business Logic	Resolved
PVE-002	Low	Revisited Pool decimals() Logic in Pool	Coding Practices	Resolved
PVE-003	Low	Improved withdrawNativeToken() Logic in WrappedNativeTokenPool	Time And State	Resolved
PVE-004	Low	Improved Caller/Input Validation in LiquidationFlashloan	Security Features	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-006	Low	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Sybil-Based Attack to Steal Vesting Reward

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: Pool
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The DeltaPrime protocol has a Pool contract which is designed to allow users to deposit and borrow asset from a dedicated user account. Depositors are rewarded with the interest rates collected from borrowers. What's more, deposits and debts are tokenized for accounting purposes. To elaborate, we show below the related transfer() routine to transfer the deposit position.

```
    function transfer(address recipient, uint256 amount) external override returns (bool
        ) {
        if(recipient == address(0)) revert TransferToZeroAddress();

        if(recipient == address(this)) revert TransferToPoolAddress();

        _accumulateDepositInterest(msg.sender);

        if(_deposited[msg.sender] < amount) revert TransferAmountExceedsBalance(amount,
            _deposited[msg.sender]);

        // (this is verified in "require" above)
        unchecked {
            _deposited[msg.sender] -= amount;
        }

        _accumulateDepositInterest(recipient);
        _deposited[recipient] += amount;

        // Handle rewards
        if(address(poolRewarder) != address(0) && amount != 0){
```

```
            uint256 unstaked = poolRewarder.withdrawFor(amount, msg.sender);
            if(unstaked > 0) {
                poolRewarder.stakeFor(unstaked, recipient);
            }
        }

        emit Transfer(msg.sender, recipient, amount);

        return true;
    }
```

Listing 3.1: `Pool::transfer()`

We notice in the above `transfer()` routine, it allows the user to transfer the deposit to another account. It comes to our attention that the deposit position may have implication in the associated `VestingDistributor` contract, which distributes the pool's spread among vesting participants. However, when the deposit position is transferred, the respective vesting implication is not taken care of. As a result, a Sybil attack may be mounted to steal the pool's spread reward.

Recommendation Properly synchronize with the `VestingDistributor` contract when a deposit position is being transferred to fairly distribute the pool's spread.

Status The issue has been fixed by this commit: `a9356c3a`.

3.2 Revisited Pool decimals() Logic in Pool

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `DeltaPrime` protocol has a `Pool` contract to tokenize the user deposits and debts. While examining the pool tokenization logic, we notice the default `decimals` may be revised to be consist with the underlying token.

To elaborate, we show below the related aspects about the pool token contract. It comes to our attention that it currently takes 0 as the decimals, which is not consistent with the underlying `tokenAddress()`. The inconsistency may bring unnecessary confusion to user wallets or other front-end client programs. With that, we suggest to ensure the consistency of the pool token's decimals with the underlying token.

```
    function name() public virtual pure returns(string memory _name){
        _name = "";
    }

    function symbol() public virtual pure returns(string memory _symbol){
        _symbol = "";
    }

    function decimals() public virtual pure returns(uint8 decimals){
        decimals = 0;
    }

    function totalSupply() public view override returns (uint256) {
        return balanceOf(address(this));
    }
```

Listing 3.2: The `Pool`'s Token Contract

Recommendation Ensure the decimals consistency of the pool token with the underlying token.

Status The issue has been fixed by this commit: `e2320d8`.

3.3 Improved withdrawNativeToken() Logic in WrappedNativeTokenPool

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WrappedNativeTokenPool`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

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` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `WrappedNativeTokenPool` as an example, the `withdrawNativeToken()` function (see the code snippet below) is provided to externally call 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 61) starts before effecting the update on internal state (line 63), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `read-only re-entrancy` on the current pool rates. Note that there may be no harm caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
48     function withdrawNativeToken(uint256 _amount) external nonReentrant {
49         if(_amount > IERC20(tokenAddress).balanceOf(address(this))) revert
               InsufficientPoolFunds();
50 
51         _accumulateDepositInterest(msg.sender);
52 
53         if(_amount > _deposited[address(this)]) revert BurnAmountExceedsBalance();
54         // verified in "require" above
55         unchecked {
56             _deposited[address(this)] -= _amount;
57         }
58         _burn(msg.sender, _amount);
59 
60         IWrappedNativeToken(tokenAddress).withdraw(_amount);
61         payable(msg.sender).safeTransferETH(_amount);
62 
63         _updateRates();
64 
65         if (address(poolRewarder) != address(0)) {
66             poolRewarder.withdrawFor(_amount, msg.sender);
67         }
68 
69         emit Withdrawal(msg.sender, _amount, block.timestamp);
70     }
```

Listing 3.3: LibLimitOrder::withdrawNativeToken()

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status The issue has been fixed by this commit: `64ee7c2`.

3.4 Improved Caller/Input Validation in LiquidationFlashloan

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LiquidationFlashloan`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `DeltaPrime` protocol has a `LiquidationFlashloan` contract to facilitate the liquidation of underwater loans. This liquidation contract makes a flashloan from the popular `AaveV3` lending protocol, which will invoke the callback to start the liquidation process. It comes to our attention that this callback is not guarded and allow any one to invoke it.

In the following, we show the code snippet from the liquidation callback routine `executeOperation()`. While it faithfully executes the intended logic in receiving the funds from the lending pool and liquidating the underwater loans. However, it needs to be guarded to ensure the caller is the intended `AaveV3` lending pool. Fortunately, this `LiquidationFlashloan` contract is a helper and does not supposed to hold any user funds. With that, we suggest to follow the best practice and validate its caller.

```
function executeOperation(
  address[] calldata,
  uint256[] calldata,
  uint256[] calldata,
  address,
  bytes calldata _params
) public override returns (bool) {
  LiqEnrichedParams memory lep = getLiqEnrichedParams(_params);

  // Use calldata instead of memory in order to avoid the "Stack Too deep" CompileError
  address[] calldata assets = getAssets();
  uint256[] calldata amounts = getAmounts();
  uint256[] calldata premiums = getPremiums();

  for (uint32 i = 0; i < assets.length; i++) {
    IERC20(assets[i]).approve(lep.loan, 0);
    IERC20(assets[i]).approve(lep.loan, amounts[i]);
  }

  (
    AssetAmount[] memory assetSurplus,
    AssetAmount[] memory assetDeficit
  ) = liquidateLoanAndGetSurplusDeficitAssets(_params, lep, assets, amounts, premiums);
```

```
    ...
}
```

Listing 3.4: `LiquidationFlashloan::executeOperation()`

Recommendation Enforce necessary caller authorization in the above `executeOperation()` routine.

Status The issue has been fixed by this commit: `8bfa3cc`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.5: USDT Token Contract

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , 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 `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.6: `SafeERC20::safeApprove()`

In current implementation, if we examine the `UniswapV2Intermediary::addLiquidity()` routine that is designed to add liquidity. To accommodate the specific idiosyncrasy, there is a need to make use of `safeApprove()` twice: the first one resets the allowance while the second one sets the intended allowance (lines 77 – 78).

```
71      function addLiquidity(address tokenA, address tokenB, uint amountA, uint amountB,
            uint amountAMin, uint amountBMin) external override nonReentrant returns (
            address, uint, uint, uint) {
72          require(amountA > 0, "amountADesired has to be greater than 0");
73          require(amountB > 0, "amountBDesired to sell has to be greater than 0");
74          require(amountAMin > 0, "amountAMin has to be greater than 0");
75          require(amountBMin > 0, "amountBMin has to be greater than 0");
76
77          tokenA.safeApprove(address(router), amountA);
```

```
        tokenB.safeApprove(address(router), amountB);

        address lpTokenAddress = getPair(tokenA, tokenB);

        require(isTokenWhitelisted[tokenA], 'Trying to LP unsupported token');
        require(isTokenWhitelisted[tokenB], 'Trying to LP unsupported token');
        require(tokenManager.isTokenAssetActive(lpTokenAddress), 'Trying to add unsupported LP token');

        uint liquidity;
        (amountA, amountB, liquidity) =
           router.addLiquidity(tokenA, tokenB, amountA, amountB, amountAMin, amountBMin, address(this), block.timestamp);

        lpTokenAddress.safeTransfer(msg.sender, IERC20Metadata(lpTokenAddress).balanceOf(address(this)));
        if (IERC20Metadata(tokenA).balanceOf(address(this)) > 0) {
            tokenA.safeTransfer(msg.sender, IERC20Metadata(tokenA).balanceOf(address(this)));
        }
        if ( IERC20Metadata(tokenB).balanceOf(address(this)) > 0) {
            tokenB.safeTransfer(msg.sender, IERC20Metadata(tokenB).balanceOf(address(this)));
        }

        return (lpTokenAddress, amountA, amountB, liquidity);
    }
```

Listing 3.7: `UniswapV2Intermediary::addLiquidity()`

Note the `LiquidationFlashloan::executeOperation()()` routine can be similarly improved.

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

Status The issue has been fixed by this commit: `8bfa3cc`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `DeltaPrime` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and contract upgrade). It 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 `owner` account and its related privileged accesses in current contracts.

```
    function addTokenAssets(Asset[] memory tokenAssets) public onlyOwner {
        for (uint256 i = 0; i < tokenAssets.length; i++) {
            _addTokenAsset(tokenAssets[i].asset, tokenAssets[i].assetAddress,
                tokenAssets[i].debtCoverage);
        }
    }

    function activateToken(address token) public onlyOwner {
        require(tokenToStatus[token] == _INACTIVE, "Must be inactive");
        tokenToStatus[token] = _ACTIVE;
        emit TokenAssetActivated(msg.sender, token, block.timestamp);
    }

    function deactivateToken(address token) public onlyOwner {
        require(tokenToStatus[token] == _ACTIVE, "Must be active");
        tokenToStatus[token] = _INACTIVE;
        emit TokenAssetDeactivated(msg.sender, token, block.timestamp);
    }

    function removeTokenAssets(bytes32[] memory _tokenAssets) public onlyOwner {
        for (uint256 i = 0; i < _tokenAssets.length; i++) {
            _removeTokenAsset(_tokenAssets[i]);
        }
    }
```

Listing 3.8: Example Privileged Operations in `TokenManager`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. Note that 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.

Moreover, it should be noted that current contracts are to be deployed behind a proxy with the typical `Diamond` implementation. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

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 mitigatd as the team clarifies the use of timelock and multisig. In future they will switch to be a `DAO`-like governance contract.

4 Conclusion

In this audit, we have analyzed the design and implementation of the DeltaPrime protocol, which is a lending platform and allows under-collateral borrowing from pooled deposits. The key mechanism is to enable fund-lending not to a personal account, but a special purpose smart-contract. The contract automatically guards solvency and every activity needs to undergo a series of checks. The insolvency risk is further mitigated by a decentralized liquidation mechanism allowing anyone to forcibly repay part of the loan due to assets price movements caused by external factors. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

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