SMART CONTRACT AUDIT REPORT

for

DeltaPrime

Prepared By: Xiaomi Huang

PeckShield
September 8, 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	September 8, 2023	Xuxian Jiang	Final Release
1.0-rc	September 2, 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 allows under-collateralized 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
Website	https://deltaprime.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 8, 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 (5e543a8)

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 (df733db)

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 the 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.

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

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 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) [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

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.

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.

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	0	
Medium	7	■■■■■■■
Low	2	■■
Undetermined	1	■
Total	10	

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 7 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 undetermined recommendation.

Table 2.1: Key DeltaPrime Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Redundant Code and Wrong Method Naming in UniswapV3IntegrationHelper	Business Logic	Resolved
PVE-002	Medium	Denial-of-Service Issue in DepositSwap	Coding Practices	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-004	Medium	Inconsistent getMultipler() Calculation in VestingDistributor	Business Logic	Resolved
PVE-005	Low	Unintended ProtocolExposureChanged Event Generation	Coding Practices	Resolved
PVE-006	Medium	Missing Mapping Verification in SmartLoansFactory	Coding Practices	Resolved
PVE-007	Medium	Incorrect Decimals For Health Calculation in HealthMeterFacetProd	Business Logic	Resolved
PVE-008	Medium	Incorrect Withdrawal Logic in RecoveryFacet	Business Logic	Resolved
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-010	Medium	Improved Caller/Input Validation in LiquidationFlashloan	Security Features	Resolved

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 Redundant Code and Wrong Method Naming in UniswapV3IntegrationHelper

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `UniswapV3IntegrationHelper`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DeltaPrime` protocol has a `UniswapV3IntegrationHelper` contract, which greatly facilitates the integration with `UniswapV3`-based DEXes. While examining the integration logic, we notice a key price-related routine should be improved.

To elaborate, we show below the related `sqrtPriceX96ToUint()` routine. As the name indicates, this routine is used to compute the `Uniswap V3` pair price from the given `Q64.96` number. Specifically, the price calculation involves two numbers, i.e., `numerator1` and `numerator2` with the purpose of having the result at the given `decimals`. However, it comes to our attention that each number is computed twice, which is redundant and brings confusion to the intended purpose. Note if there is a need to compute a `Uniswap V3` pool token price, we will need the first computation.

```
    function sqrtPriceX96ToUint(uint160 sqrtPriceX96, uint8 decimalsToken0)
    internal
    view //TODO: pure
    returns (uint256)
    {
        {
            uint256 numerator1 = uint256(sqrtPriceX96) * uint256(sqrtPriceX96);
            uint256 numerator2 = 10**decimalsToken0;
        }

        uint256 numerator1 = uint256(sqrtPriceX96);
        uint256 numerator2 = 10**decimalsToken0;
```

```
        return FullMath.mulDiv(numerator1, numerator2, 2 ** 96);
    }
```

Listing 3.1: UniswapV3IntegrationHelper::sqrtPriceX96ToUint()

Recommendation Revise the above sqrtPriceX96ToUint() routine to compute the intended pool token price.

Status The issue has been fixed by this commit: f926ecc.

3.2 Denial-of-Service Issue in DepositSwap

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: DepositSwap
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The DeltaPrime protocol manages the user deposits and borrows with a Pool contract. It also provides a DepositSwap contract to facilitate the token withdrawal, swap and deposit. While examining the streamlined token withdrawal and deposit logic, we notice it suffers from a possible denial-of-service issue.

To elaborate, we show below the related _withdrawFromPool() function. It comes to our attention that it imposes the following two requirements: require(pool.balanceOf(address(this))== 0) (line 57) and require(token.balanceOf(address(this))== 0) (line 58). As a result, any dust donation will fail these two requirements, hence making this DepositSwap contract nonfunctional.

```
53      function _withdrawFromPool(Pool pool, IERC20 token, uint256 amount, address user)
            private {
54          uint256 userInitialFromTokenDepositBalance = pool.balanceOf(user);
55
56          require(userInitialFromTokenDepositBalance >= amount, "Insufficient fromToken
                deposit balance");
57          require(pool.balanceOf(address(this)) == 0, "Contract initial deposit balance
                should be 0");
58          require(token.balanceOf(address(this)) == 0, "Contract initial fromToken balance
                 must be 0");
59
60          pool.transferFrom(user, address(this), amount);
61          require(pool.balanceOf(address(this)) == amount, "amountFromToken and post-
                transfer contract balance mismatch");
62          require(pool.balanceOf(user) == userInitialFromTokenDepositBalance - amount, "
                user post-transfer balance is incorrect");
```

```
        pool.withdraw(amount);

        require(pool.balanceOf(address(this)) == 0, "Post-withdrawal contract deposit
            balance must be 0");
        require(token.balanceOf(address(this)) == amount, "Post-withdrawal contract
            fromToken balance is incorrect");
    }
```

Listing 3.2: `DepositSwap::_withdrawFromPool()`

Recommendation Revise the `DepositSwap` contract to avoid the above-mentioned denial-of-service issue. Note both deposit and withdrawal logic share the same issue.

Status The issue has been fixed by this commit: `d3d16e1`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- 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 3.3: 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 3.4: `SafeERC20::safeApprove()`

In current implementation, if we examine the `DepositSwap::_yakSwap()` routine that is designed to swap tokens. 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 (line 89).

```
88      function _yakSwap(address[] calldata path, address[] calldata adapters, uint256
            amountIn, uint256 amountOut) private {
89          IERC20(path[0]).approve(YY_ROUTER, amountIn);
90
```

```
        IYieldYakRouter router = IYieldYakRouter(YY_ROUTER);


        IYieldYakRouter.Trade memory trade = IYieldYakRouter.Trade({
            amountIn: amountIn,
            amountOut: amountOut,
            path: path,
            adapters: adapters
        });

        router.swapNoSplit(trade, address(this), 0);
    }
```

Listing 3.5: `DepositSwap::_yakSwap()`

Note the `PoolRewarder::getRewardsFor()` routine can be similarly improved.

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

Status The issue has been fixed by this commit: `9425321`.

3.4 Inconsistent getMultipler() Calculation in VestingDistributor

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VestingDistributor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DeltaPrime` protocol has a `VestingDistributor` contract, which is used to distribute the pool's spread among vesting participants. The spread distribution may be affected by the participants' multiplier and the multipler calculation is flawed.

To elaborate, we show below the related `getMultiplier()` routine. It has a rather straightforward logic in computing the multiplier based on the vesting duration. However, it comes to our attention that if the vesting duration is within a single day, the multiplier `1e18` needs to be returned and the current implementation returns `1`.

```
    function getMultiplier(uint256 time) public pure returns (uint256){
        if (time >= 30 * ONE_DAY) return 2e18; // min. 30 days
        if (time >= 29 * ONE_DAY) return 1.99e18; // min. 29 days
        if (time >= 28 * ONE_DAY) return 1.98e18; // min. 28 days
        ...
        if (time >= 4 * ONE_DAY) return 1.468e18; // min. 4 days
```

```
        if (time >= 3 * ONE_DAY) return 1.4e18; // min. 3 days
        if (time >= 2 * ONE_DAY) return 1.32e18; // min. 2 days
        if (time >= 1 * ONE_DAY) return 1.2e18; // min. 1 day

        return 1;
    }
```

Listing 3.6: `VestingDistributor::getMultiplier()`

Recommendation Fix the above `getMultiplier()` routine to compute the correct multipler.

Status The issue has been fixed by this commit: `3eb730d`.

3.5 Unintended ProtocolExposureChanged Event Generation

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `TokenManager` contract as an example. This contract has public functions that are used to update the protocol exposure. While examining the events that reflect the exposure changes, we notice the emitted important events do not correctly reflect important state changes. Specifically, when the exposure is increased/decreased, the emitted event needs to include the updated exposure after the change (lines 106 and 116).

```
98    function increaseProtocolExposure(bytes32 assetIdentifier, uint256 exposureIncrease)
          public onlyPrimeAccountOrOwner {
99        bytes32 group = identifierToExposureGroup[assetIdentifier];
100       if(group != ""){
101           Exposure storage exposure = groupToExposure[group];
102           if(exposure.max != 0){
103               exposure.current += exposureIncrease;
104               require(exposure.current <= exposure.max, "Max asset exposure breached")
                  ;
105               emit ProtocolExposureChanged(msg.sender, group, exposureIncrease, block.
                  timestamp);
```

```
            }
        }
    }

    function decreaseProtocolExposure(bytes32 assetIdentifier, uint256 exposureDecrease) public onlyPrimeAccountOrOwner {
        bytes32 group = identifierToExposureGroup[assetIdentifier];
        if(group != ""){
            Exposure storage exposure = groupToExposure[group];
            if(exposure.max != 0){
                exposure.current = exposure.current <= exposureDecrease ? 0 : exposure.current - exposureDecrease;
                emit ProtocolExposureChanged(msg.sender, group, exposureDecrease, block.timestamp);
            }
        }
    }
```

Listing 3.7: `TokenManager::increaseProtocolExposure()/decreaseProtocolExposure()`

Recommendation Properly emit respective events when the protocol exposure is updated.

Status This issue has been fixed in the following commit: `500ad70`.

3.6 Missing Mapping Verification in SmartLoansFactory

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SmartLoansFactory`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `DeltaPrime` protocol makes use of a `SmartLoansFactory` contract to instantiate user loan contracts. It also provides a convenient function `createAndFundLoan()` to create a new loan contract and fund the loan in one single call. In the process of analyzing this function, we notice it can be improved.

To elaborate, we show below the related `createAndFundLoan()` function, which simply creates a new loan contract and provides fund to it. However, it does not properly validate the given user input. Specifically, the given `_fundedAsset` may not be mapped to the given `_assetAddress`. And the function correction assumes the given input is consistent.

```
    function createAndFundLoan(bytes32 _fundedAsset, address _assetAddress, uint256 _amount) public virtual hasNoLoan returns (SmartLoanDiamondBeacon) {
        SmartLoanDiamondProxy beaconProxy = new SmartLoanDiamondProxy(payable(address(smartLoanDiamond)),
```

```
            abi.encodeWithSelector(SmartLoanViewFacet.initialize.selector, msg.sender)
        );
        SmartLoanDiamondBeacon smartLoan = SmartLoanDiamondBeacon(payable(address(
            beaconProxy)));

        //Fund account with own funds and credit
        IERC20Metadata token = IERC20Metadata(_assetAddress);
        address(token).safeTransferFrom(msg.sender, address(this), _amount);
        address(token).safeApprove(address(smartLoan), _amount);

        //Update registry and emit event
        updateRegistry(address(smartLoan), msg.sender);

        (bool success, bytes memory result) = address(smartLoan).call(abi.
            encodeWithSelector(AssetsOperationsFacet.fund.selector, _fundedAsset,
            _amount));
        ProxyConnector._prepareReturnValue(success, result);

        emit SmartLoanCreated(address(smartLoan), msg.sender, _fundedAsset, _amount);

        return smartLoan;
    }
```

Listing 3.8: `SmartLoansFactory::createAndFundLoan()`

Recommendation Validate the given input to the above function is consistent and expected.

Status The issue has been fixed by this commit: `4380cb4`.

3.7 Incorrect Decimals For Health Calculation in HealthMeterFacetProd

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HealthMeterFacetProd`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Each `HealthMeterFacetProd` contract in `DeltaPrime` is used to compute the health rate for a given loan contract. Our analysis on the health rate calculation shows it needs to be improved.

To elaborate, we show below its implementation of one key routine `getHealthMeter()`. This function is designed to compute current health meter associated with the loan. However, the debt calculation in `borrowed` wrongfully assumes the token has the same 18 decimals (line 100) and thus may yield an incorrect result. In other words, the debt computation needs to take the following approach: `borrowed += (ownedAssetsPrices[i].price * pool.getBorrowed(address(this))1e10/(10**token.decimals()))`.

```
    function getHealthMeter() public view returns (uint256) {
        AssetPrice[] memory ownedAssetsPrices = _getOwnedAssetsWithNativePrices();

        bytes32 nativeTokenSymbol = DeploymentConstants.getNativeTokenSymbol();
        ITokenManager tokenManager = DeploymentConstants.getTokenManager();

        uint256 weightedCollateral;
        uint256 weightedCollateralPlus = ownedAssetsPrices[0].price * address(this).balance * tokenManager.debtCoverage(tokenManager.getAssetAddress(nativeTokenSymbol, true)) / (10 ** 26);
        uint256 weightedCollateralMinus = 0;
        uint256 weightedBorrowed = 0;
        uint256 borrowed = 0;

        for (uint256 i = 0; i < ownedAssetsPrices.length; i++) {
            Pool pool;
            try tokenManager.getPoolAddress(ownedAssetsPrices[i].asset) returns (address poolAddress) {
                pool = Pool(poolAddress);
            } catch {
                continue;
            }
            IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssetsPrices[i].asset, true));
            uint256 _balance = token.balanceOf(address(this));
            uint256 _borrowed = pool.getBorrowed(address(this));
            if (_balance > _borrowed) {
                weightedCollateralPlus = weightedCollateralPlus + (ownedAssetsPrices[i].price * (_balance - _borrowed) * tokenManager.debtCoverage(address(token)) / (10 ** token.decimals() * 1e8));
            } else {
                weightedCollateralMinus = weightedCollateralMinus + (ownedAssetsPrices[i].price * (_borrowed - _balance) * tokenManager.debtCoverage(address(token)) / (10 ** token.decimals() * 1e8));
            }
            weightedBorrowed = weightedBorrowed + (ownedAssetsPrices[i].price * pool.getBorrowed(address(this)) * tokenManager.debtCoverage(address(token)) / (10 ** token.decimals() * 1e8));
            borrowed = borrowed + (ownedAssetsPrices[i].price * pool.getBorrowed(address(this)) / 1e8);
        }
        if (weightedCollateralPlus > weightedCollateralMinus) {
            weightedCollateral = weightedCollateralPlus - weightedCollateralMinus;
        }

        uint256 multiplier = 100 * 1e18; // 18 decimal points

        if (borrowed == 0) return multiplier;
```

```
        if (weightedCollateral > 0 && weightedCollateral + weightedBorrowed > borrowed)
            {
            return (weightedCollateral + weightedBorrowed - borrowed) * multiplier /
                weightedCollateral;
        }

        return 0;
    }
```

Listing 3.9: `HealthMeterFacetProd::getHealthMeter()`

Recommendation Correct the above routine to properly compute a loan's health rate.

Status The issue has been fixed by this commit: `221b130`.

3.8 Incorrect Withdrawal Logic in RecoveryFacet

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `RecoveryFacet`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DeltaPrime` protocol has a `RecoveryFacet` contract to recovery funds in emergency situations. However, our analysis shows that the current withdrawal logic is incorrectly implemented.

In the following, we show its implementation of a key `_withdraw()` routine. The withdrawal logic checks the current asset and the issue comes from the handling of withdrawing a staked position. Specifically, when the given asset is part of a staked position, it will replace the withdrawn one with the last available one in the current position list. However, when it adjusts the associated protocol exposure, it uses the decimals of the existing token, not the replaced one. As a result, the protocol exposure logic may be messed up.

```
    function _withdraw(bytes32 _asset) internal returns (uint256 _amount) {
        ITokenManager tokenManager = DeploymentConstants.getTokenManager();

        if (
            _asset == "VF_USDC_MAIN_AUTO" ||
            _asset == "VF_USDT_MAIN_AUTO" ||
            _asset == "VF_AVAX_SAVAX_AUTO" ||
            _asset == "VF_SAVAX_MAIN_AUTO"
        ) {
            IStakingPositions.StakedPosition[] storage positions = DiamondStorageLib
```

```
                .stakedPositions();
            uint256 positionsLength = positions.length;
            for (uint256 i; i != positionsLength; ++i) {
                IStakingPositions.StakedPosition memory position = positions[i];
                if (position.identifier != _asset) continue;

                positions[i] = positions[positionsLength - 1];
                positions.pop();

                IVectorFinanceCompounder compounder = _getAssetPoolHelper(
                    position.asset
                ).compounder();
                uint256 shares = compounder.balanceOf(address(this));
                uint256 stakedBalance = compounder.getDepositTokensForShares(shares);

                _amount = compounder.depositTracking(address(this));
                address(compounder).safeTransfer(msg.sender, _amount);

                uint256 decimals = IERC20Metadata(tokenManager.getAssetAddress(positions
                    [i].symbol, true)).decimals();
                tokenManager.decreaseProtocolExposure(positions[i].identifier,
                    stakedBalance * 1e18 / 10**decimals);

                break;
            }
        }
        ...
    }
```

Listing 3.10: `RecoveryFacet::_withdraw()`

Recommendation Revisit the above logic to make use of the correct decimals for protocol exposure adjustment.

Status The issue has been fixed by this commit: `fd44a51`.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- 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 3.11: 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.

3.10 Improved Caller/Input Validation in LiquidationFlashloan

- ID: PVE-010
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `LiquidationFlashloan`
- Category: Security Features [4]
- 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 as well as the input arguments.

```
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 3.12: `LiquidationFlashloan::executeOperation()`

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

Status The issue has been fixed by this commit: `217bd23`.

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-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.