SMART CONTRACT AUDIT REPORT

for

DeltaPrime Protocol

Prepared By: Yiqun Chen

Hangzhou, China

February 12, 2022

Document Properties

Client	Delta Prime Labs
Title	Smart Contract Audit Report
Target	DeltaPrime Protocol
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 12, 2022	Jing Wang	Final Release
1.0-rc	January 17, 2022	Jing Wang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the `DeltaPrime Protocol` design document and related smart contract source code, 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 branch of `DeltaPrime Protocol` 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 Protocol

`DeltaPrime Protocol` 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 `DeltaPrime Protocol` is as follows:

Table 1.1: Basic Information of DeltaPrime Protocol

Item	Description
Name	Delta Prime Labs
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 12, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that `DeltaPrime Protocol` 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 (7a85d6e)

And this 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 (0eed48a)

Also, due to the `DeltaPrime Protocol` is using the `redstone-evm-connector` project to provide oracle data by the meta-transaction pattern. So this audit also includes the following single file which is used to parse and valid the data provided by authorized signer. Note that this pattern is working on the assumption that selected signer is trusted and how the signer is selected is not part of this audit.

- https://github.com/redstone-evm-connectorcontracts/PriceAwareUpgradeable.sol (00e81e5)

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 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

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

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.

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 `DeltaPrime Protocol` implementation. 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	1	■
Low	2	■ ■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, 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, 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key DeltaPrime Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Informational	Lack of Emitting Events	Recommendation	Fixed
PVE-004	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed

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 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SmartLoan
- Category: Time and State [7]
- CWE subcategory: CWE-682 [4]

Description

The SmartLoan contract has a helper routine, i.e., closeLoan(), that is designed to allow the user to sell all of the assets. It has a rather straightforward logic to swap assets to AVAX by calling the sellAsset() routine to actually perform the intended token swap.

```
  function closeLoan() external payable onlyOwner nonReentrant remainsSolvent {
      bytes32[] memory assets = exchange.getAllAssets();
      for (uint256 i = 0; i < assets.length; i++) {
        uint256 balance = getERC20TokenInstance(assets[i]).balanceOf(address(this));
        if (balance > 0) {
          sellAsset(assets[i], balance, 0);
        }
      }

      uint256 debt = getDebt();
      require(address(this).balance >= debt, "Selling out all assets without repaying the whole debt is not allowed");
      repay(debt);
      emit LoanClosed(debt, address(this).balance, block.timestamp);

      uint256 balance = address(this).balance;
      if (balance > 0) {
        payable(msg.sender).safeTransferETH(balance);
        emit Withdrawn(msg.sender, balance, block.timestamp);
      }
    }
```

Listing 3.1: SmartLoan::closeLoan()

```
    function sellAsset(bytes32 asset, uint256 _amount, uint256 _minAvaxOut) private {
        IERC20Metadata token = getERC20TokenInstance(asset);
        address(token).safeTransfer(address(exchange), _amount);
        exchange.sellAsset(asset, _amount, _minAvaxOut);
    }
```

Listing 3.2: `SmartLoan::sellAsset()`

```
    function sellAsset(bytes32 _token, uint256 _exactERC20AmountIn, uint256
        _minAvaxAmountOut) external override nonReentrant returns (bool) {
        require(_exactERC20AmountIn > 0, "Amount of tokens to sell has to be greater
            than 0");

        address tokenAddress = getAssetAddress(_token);
        IERC20 token = IERC20(tokenAddress);
        token.approve(address(pangolinRouter), _exactERC20AmountIn);

        (bool success, ) = address(pangolinRouter).call{value: 0}(
          abi.encodeWithSignature("swapExactTokensForAVAX(uint256,uint256,address[],
              address,uint256)", _exactERC20AmountIn, _minAvaxAmountOut,
              getPathForTokenToAVAX(tokenAddress), msg.sender, block.timestamp)
        );

        if (!success) {
          address(token).safeTransfer(msg.sender, token.balanceOf(address(this)));
          return false;
        }
        payable(msg.sender).safeTransferETH(address(this).balance);
        emit TokenSell(msg.sender, _exactERC20AmountIn, block.timestamp, success);
        return true;
    }
```

Listing 3.3: `PangolinExchange::sellAsset()`

To elaborate, we show above the related routines. We notice the token swap is routed to `pangolinRouter` and the actual swap operation `swapExactTokensForAVAX()` does not specify any restriction (with `_minAvaxOut=0`) on possible slippage when calling from the `closeLoan()` routine and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Note another routine `sellAssetForTargetAvax()` shares the same issue.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the teams. And the team clarifies that, by design, if the users want to control the slippage, they need to sell the assets one by one.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Pool`
- 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., configure `_borrowerRegistry` and pause protocol). 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 contract.

To elaborate, we show below the function provided to configure the `borrowersRegistry_` contract, which stores the critical information about whether an account can borrow funds.

```
    function setBorrowersRegistry(IBorrowersRegistry borrowersRegistry_) external
        onlyOwner {
        require(address(borrowersRegistry_) != address(0), "The borrowers registry
            cannot set to a null address");
        require(AddressUpgradeable.isContract(address(borrowersRegistry_)), "Must be a
            contract");

        _borrowersRegistry = borrowersRegistry_;
    }
```

Listing 3.4: `Pool::setBorrowersRegistry()`

```
    modifier canBorrow() {
        require(address(_borrowersRegistry) != address(0), "Borrowers registry is not
            configured");
        require(_borrowersRegistry.canBorrow(msg.sender), "Only the accounts authorised
            by borrowers registry may borrow");
        require(totalSupply() != 0, "Cannot borrow from an empty pool");
```

```
        _;
        require((totalBorrowed() * 1e18) / totalSupply() <=
            MAX_POOL_UTILISATION_FOR_BORROWING, "The pool utilisation cannot be greater
            than 95%");
    }
```

Listing 3.5: `Pool::canBorrow()`

Also, the `owner` could pause the `deposit()` and `withdraw()` functions of the pool by setting the `ratesCalculator_` to `address(0)`, which is checked on `_updateRates()` when depositing and withdrawing from the pool.

```
    function setRatesCalculator(IRatesCalculator ratesCalculator_) external onlyOwner {
        // setting address(0) ratesCalculator_ freezes the pool
        require(AddressUpgradeable.isContract(address(ratesCalculator_))  address(
            ratesCalculator_) == address(0), "Must be a contract");
        _ratesCalculator = ratesCalculator_;
        if (address(ratesCalculator_) != address(0)) {
          _updateRates();
        }
      }
```

Listing 3.6: `Pool::setRatesCalculator()`

```
    function _updateRates() internal {
        require(address(_ratesCalculator) != address(0), "Pool is frozen: cannot perform
             deposit, withdraw, borrow and repay operations");
        depositIndex.setRate(_ratesCalculator.calculateDepositRate(totalBorrowed(),
            totalSupply()));
        borrowIndex.setRate(_ratesCalculator.calculateBorrowingRate(totalBorrowed(),
            totalSupply()));
    }
```

Listing 3.7: `Pool::_updateRates()`

```
    /**
    * Deposits the message value
    * It updates user deposited balance, total deposited and rates
    **/
   function deposit() external payable virtual nonReentrant {
     _accumulateDepositInterest(msg.sender);

     _mint(msg.sender, msg.value);
     _updateRates();

     emit Deposit(msg.sender, msg.value, block.timestamp);
   }

   /**
    * Withdraws selected amount from the user deposits
    * @dev _amount the amount to be withdrawn
    **/
```

```
  function withdraw(uint256 _amount) external nonReentrant {
    require(address(this).balance >= _amount, "There is not enough funds in the pool to
        fund the loan");

    _accumulateDepositInterest(msg.sender);

    _burn(msg.sender, _amount);

    payable(msg.sender).safeTransferETH(_amount);

    _updateRates();

    emit Withdrawal(msg.sender, _amount, block.timestamp);
  }
```

Listing 3.8: `Pool::deposit()and withdraw()`

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.

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 The issue has been confirmed by the team. The team clarifies that a timelock contract is used as the `owner`.

3.3 Lack of Emitting Meaningful Events

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Pool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

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.

As mentioned in Section 3.2, the `borrowersRegistry_` contract stores the critical information about whether an account can borrow funds. While examining the events that reflect the `borrowersRegistry_` contract changes, we notice there is a lack of emitting related events that reflect important state changes. Specifically, when the `borrowersRegistry_` contract is being updated, there is no respective event being emitted to reflect the change (line 77).

```
73     function setBorrowersRegistry(IBorrowersRegistry borrowersRegistry_) external onlyOwner {
74         require(address(borrowersRegistry_) != address(0), "The borrowers registry cannot set to a null address");
75         require(AddressUpgradeable.isContract(address(borrowersRegistry_)), "Must be a contract");
76 
77         _borrowersRegistry = borrowersRegistry_;
78     }
```

Listing 3.9: `Pool::setBorrowersRegistry()`

Recommendation Properly emit the related `borrowersRegistry_` change event when the `borrowersRegistry_` is being updated.

Status The issue has been fixed by this commit: `0eed48a`.

3.4 Accommodation of approve() Idiosyncrasies

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PangolinExchange`
- 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 3.10: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `PangolinExchange::sellAsset()` routine as an example. This routine is designed to approve the `pangolinRouter` contract to swap `_token` into `AVAX`. To accommodate the specific idiosyncrasy, for each `approve()` (line 71), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Also, the `IERC20` interface has defined the `approve()` interface with a `bool` return value, but the above implementation does not have the return value. As a result, a normal `IERC20`-based `approve()` with a non-compliant token may unfortunately revert the transaction. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of current `approve()` (line 71).

```
60  /**
61  * Sells selected ERC20 token for AVAX
62  * @dev _token ERC20 token's address
63  * @dev _exactERC20AmountIn amount of the ERC20 token to be sold
64  * @dev _minAvaxAmountOut minimum amount of the AVAX token to be bought
65  **/
66  function sellAsset(bytes32 _token, uint256 _exactERC20AmountIn, uint256
        _minAvaxAmountOut) external override nonReentrant returns (bool) {
67    require(_exactERC20AmountIn > 0, "Amount of tokens to sell has to be greater than
          0");

69    address tokenAddress = getAssetAddress(_token);
70    IERC20 token = IERC20(tokenAddress);
71    token.approve(address(pangolinRouter), _exactERC20AmountIn);

73    (bool success, ) = address(pangolinRouter).call{value: 0}(
```

```
    abi.encodeWithSignature("swapExactTokensForAVAX(uint256,uint256,address[],
        address,uint256)", _exactERC20AmountIn, _minAvaxAmountOut,
        getPathForTokenToAVAX(tokenAddress), msg.sender, block.timestamp)
  );

  if (!success) {
    address(token).safeTransfer(msg.sender, token.balanceOf(address(this)));
    return false;
  }
  payable(msg.sender).safeTransferETH(address(this).balance);
  emit TokenSell(msg.sender, _exactERC20AmountIn, block.timestamp, success);
  return true;
}
```

Listing 3.11: `PangolinExchange::sellAsset()`

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

Status The issue has been fixed by this commit: `0eed48a`.

4 Conclusion

In this audit, we have analyzed the DeltaPrime Protocol design and implementation. DeltaPrime Protocol is a lending platform on Avalanche that will allow under-collateral borrowing by a smart contract from pooled deposits. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.