SMART CONTRACT AUDIT REPORT

for

INIT Capital

Prepared By: Xiaomi Huang

PeckShield

January 10, 2024

Document Properties

Client	INIT Capital
Title	Smart Contract Audit Report
Target	INIT Capital
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Jonathan Zhao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 10, 2024	Xuxian Jiang	Final Release
1.0-rc	November 21, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

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

Contents

1 Introduction **4**
- 1.1 About INIT Capital . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings **9**
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results **11**
- 3.1 Bogus Collateralization With Evil Pools . . . 11
- 3.2 Trust Issue of Admin Keys . . . 12
- 3.3 Improved Risk Parameter Enforcement in InitCore . . . 14
- 3.4 Revisited getPrice() Logic in PythOracleReader . . . 15
- 3.5 Accommodation of Non-ERC20-Compliant Tokens . . . 16

4 Conclusion **19**

References **20**

1 Introduction

Given the opportunity to review the design document and related source code of the `INIT Capital` 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 INIT Capital

`INIT Capital` is a composable `liquidity hook` money market that allows any `DeFi` protocol to permissionlessly build `Liquidity Hook` plugins and borrow liquidity to execute various `DeFi` strategies from simple to complex leverage strategies. Additionally, end users on `INIT Capital` have access to all `Hooks`, which are yield generating strategies, in a few clicks without having to use and manage many accounts and positions on multiple `DeFi` applications. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The INIT Capital

Item	Description
Name	INIT Capital
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 10, 2024

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

- https://github.com/init-capital/init-audit.git (3399c73)

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

- https://github.com/init-capital/init-audit.git (1d9d243)

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

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

1.3 Methodology

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

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be 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

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Total	5	

We have so far identified a list of potential issues. 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 high-severity vulnerability, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key INIT Capital Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Bogus Collateralization With Evil Pools	Business Logic	Resolved
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Low	Improved Risk Parameter Enforcement in InitCore	Coding Practices	Resolved
PVE-004	Low	Revisited getPrice() Logic in PythOracleReader	Business Logic	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Bogus Collateralization With Evil Pools

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: InitCore
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The INIT Capital protocol has the built-in lending functionality and allows users to add additional collateral into existing positions. While examining the collateral-adding logic, we notice an issue that requires stricter validation on the given user input.

In the following, we show the implementation of the related collateralize() routine. As the name indicates, this routine is designed to add more collateral into a given position. While it does validate the pool mode as well as the given collateral pool, we notice the collateral token validation needs to be performed on the given _pool argument, not the query result from ILendingPool(_pool).underlyingToken() (line 193). As a result, a malicious actor may create an evil pool to bypass the validity check and create a fake top-up issue.

```
185     function collateralize(uint _posId, address _pool) public virtual onlyPosOwner(
            _posId) nonReentrant {
186         IConfig _config = IConfig(config);
187         // check mode status
188         uint16 mode = _getPositionMode(_posId);
189         _require(_config.getModeStatus(mode).canCollateralize, Errors.
                COLLATERALIZE_PAUSED);
190         // check if the position mode supports _pool
191         _require(_config.isAllowedForCollateral(mode, ILendingPool(_pool).
                underlyingToken()), Errors.INVALID_MODE);
192         // update collateral on the position
193         uint amtColl = IPositionManager(POS_MANAGER).addCollateral(_posId, _pool);
194
```

```
        emit Collateralize(_posId, _pool, amtColl);
    }
```

Listing 3.1: `InitCore::collateralize()`

Recommendation Revisit the above routine to properly validate the user input. Note this issue also affects another routine, i.e., `setPositionMode()`.

Status This issue has been fixed by the following commit: `f042d63`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `INIT Capital` protocol, there is a privileged administrative account (with the `ADMIN` role). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Config` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setModeStatus(uint16 _mode, ModeStatus calldata _status) external onlyAdmin
         {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigurations[_mode].status = _status;
        emit SetModeStatus(_mode, _status);
    }

    function setMinBorrowUSD(uint16 _mode, uint128 _minBorrowUSD) external onlyAdmin {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigurations[_mode].minBorrowUSD = _minBorrowUSD;
        emit SetMinBorrowUSD(_mode, _minBorrowUSD);
    }

    function setMaxBorrowUSD(uint16 _mode, uint128 _maxBorrowUSD) external onlyAdmin {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigurations[_mode].maxBorrowUSD = _maxBorrowUSD;
        emit SetMaxBorrowUSD(_mode, _maxBorrowUSD);
    }

    function setTargetHealthAfterLiquidation_e18(
        uint16 _mode,
        uint64 _targetHealthAfterLiquidation_e18
    ) external onlyAdmin {
        _require(_mode != 0, Errors.INVALID_MODE);
        _require(_targetHealthAfterLiquidation_e18 > ONE_E18, Errors.INPUT_TOO_LOW);
        __modeConfigurations[_mode].targetHealthAfterLiquidation_e18 =
            _targetHealthAfterLiquidation_e18;
        emit SetTargetHealthAfterLiquidation_e18(_mode,
            _targetHealthAfterLiquidation_e18);
    }

    function setFlashInfo(FlashInfo calldata _info) external onlyAdmin {
        _setFlashInfo(_info);
    }

    function setWhitelistedWLps(address[] calldata _wLps, bool _status) external
        onlyAdmin {
        for (uint i; i < _wLps.length; i.uinc()) {
            whitelistedWLps[_wLps[i]] = _status;
        }
        emit SetWhitelistedWLps(_wLps, _status);
    }
```

Listing 3.2: Example Privileged Operations in `Config`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative 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 changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated with the plan to transfer the privileged account to a `multi-sig` account.

3.3 Improved Risk Parameter Enforcement in InitCore

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `InitCore`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `INIT Capital` protocol is no exception. Specifically, if we examine the `Config` contract, it has defined a number of protocol-wide risk parameters, such as `minBorrowUSD` and `maxBorrowUSD`. Our analysis shows that their enforcement needs to properly applied.

In the following, we show the implementation of the related setters `setMinBorrowUSD()` and `setMaxBorrowUSD()`. As the name indicates, they are used to configure `minBorrowUSD` and `maxBorrowUSD`. However, there enforcement is currently missing in the `InitCore` contract.

```
    function setMinBorrowUSD(uint16 _mode, uint128 _minBorrowUSD) external onlyAdmin {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigurations[_mode].minBorrowUSD = _minBorrowUSD;
        emit SetMinBorrowUSD(_mode, _minBorrowUSD);
    }

    function setMaxBorrowUSD(uint16 _mode, uint128 _maxBorrowUSD) external onlyAdmin {
        _require(_mode != 0, Errors.INVALID_MODE);
        __modeConfigurations[_mode].maxBorrowUSD = _maxBorrowUSD;
        emit SetMaxBorrowUSD(_mode, _maxBorrowUSD);
    }
```

Listing 3.3: `Config::setMinBorrowUSD()/setMaxBorrowUSD()`

```
    function borrow(
        address _pool,
        uint _amt,
        uint _posId,
        address _to
    ) public virtual onlyPosOwner(_posId) ensurePositionHealth(_posId) nonReentrant
        returns (uint shares) {
        IConfig _config = IConfig(config);
        // check pool and mode status
        PoolConfiguration memory poolConfig = _config.getPoolConfiguration(_pool);
        uint16 mode = _getPositionMode(_posId);
        _require(poolConfig.canBorrow && _config.getModeStatus(mode).canBorrow, Errors.
            BORROW_PAUSED);
        // check if the position mode supports _pool's underlying token
        _require(_config.isAllowedForBorrow(mode, ILendingPool(_pool).underlyingToken())
            , Errors.INVALID_MODE);
        // call borrow from the pool with target _to
        shares = ILendingPool(_pool).borrow(_to, _amt);
        // update debt on the position
        IPositionManager(POS_MANAGER).updatePositionDebtShares(_posId, _pool, shares.
            toInt256());

        emit Borrow(_pool, _posId, _to, _amt, shares);
    }
```

Listing 3.4: `InitCore::borrow()`

Recommendation Revise the above routine to properly enforce the protocol-wide risk parameters. Note the enforcement of `supplyCap` and `borrowCap` can be similarly improved.

Status This issue has been resolved as the team removed the min/max borrow usd.

3.4 Revisited getPrice() Logic in PythOracleReader

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `PythOracleReader`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `INIT Capital` protocol has an oracle contract `PythOracleReader` to manage and query `Pyth` oracles. In the process of analyzing the query logic, we notice the current implementation may be improved.

In the following, we show the implementation of the `getPrice()` routine. This routine is designed to return the value of the given input as native token per unit. We notice the associated configuration may require to check the stale time as well as the allowed maximum price deviation. However, it comes to our attention that current logic only checks one requirement (line 85), not both. Note that both requirements should be met to ensure the queried prices are considered valid.

```
80      function getPrice(address _token) external view returns (uint price_e36) {
81          // load and check
82          bytes32 priceId = priceIds[_token];
83          PythConfig memory config = pythConfigs[_token];
84          _require(priceId != bytes32(0), Errors.NO_PRICE_ID);
85          _require(config.maxStaleTime != 0  config.maxConfDeviation_e18 != 0, Errors.
                PYTH_CONFIG_NOT_SET);
86
87          // NOTE:
        // price: Price
        // conf: Confidence interval around the price
        // expo: Price exponent e.g. 10^8 -> expo = -8
        // publishTime: Unix timestamp describing when the price was published
        (int64 price, uint64 conf, int32 expo, uint64 publishTime) = IPyth(pyth).
            getPriceUnsafe(priceId);

        // check if the last updated is not longer than the max stale time
        _require(block.timestamp - publishTime <= config.maxStaleTime, Errors.
            MAX_STALETIME_EXCEEDED);

        // validate conf
        uint priceInUint = int(price).toUint256();
        _require(Math.mulDiv(conf, ONE_E18, priceInUint) <= config.maxConfDeviation_e18,
             Errors.CONFIDENCE_TOO_HIGH);

        // return as [USD_e36 per wei unit]
        price_e36 = priceInUint * 10 ** (36 - IERC20Metadata(_token).decimals() - uint(
            int(-expo)));
    }
```

Listing 3.5: `PythOracleReader::getPrice()`

Recommendation Revise the above routine to ensure both above-mentioned conditions are met.

Status This issue has been fixed by the following commit: `aaf88ea`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- 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 3.6: 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 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `WrapCenter::_approve()` routine that is designed to approve the spending allowance to the given spender. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 51).

```
    function _approve(address _token, address _spender, uint _amount) internal {
        if (IERC20(_token).allowance(address(this), _spender) < _amount) {
            IERC20(_token).approve(_spender, type(uint).max);
        }
    }
```

Listing 3.8: `WrapCenter::_approve()`

Note the `LendingPool::initialize()` and `InitCore::repay()` routines can be similarly improved.

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

Status This issue has been fixed by following the above suggestion.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `INIT Capital` protocol, which is `liquidity hook` money market, a composable money market that allows any `DeFi` protocol to permissionlessly build `Liquidity Hook` plugins and borrow liquidity to execute various `DeFi` strategies from simple to complex leverage strategies. Additionally, end users on `INIT Capital` have access to all `Hooks`, which are yield generating strategies, in a few clicks without having to use and manage many accounts and positions on multiple `DeFi` applications. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-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.