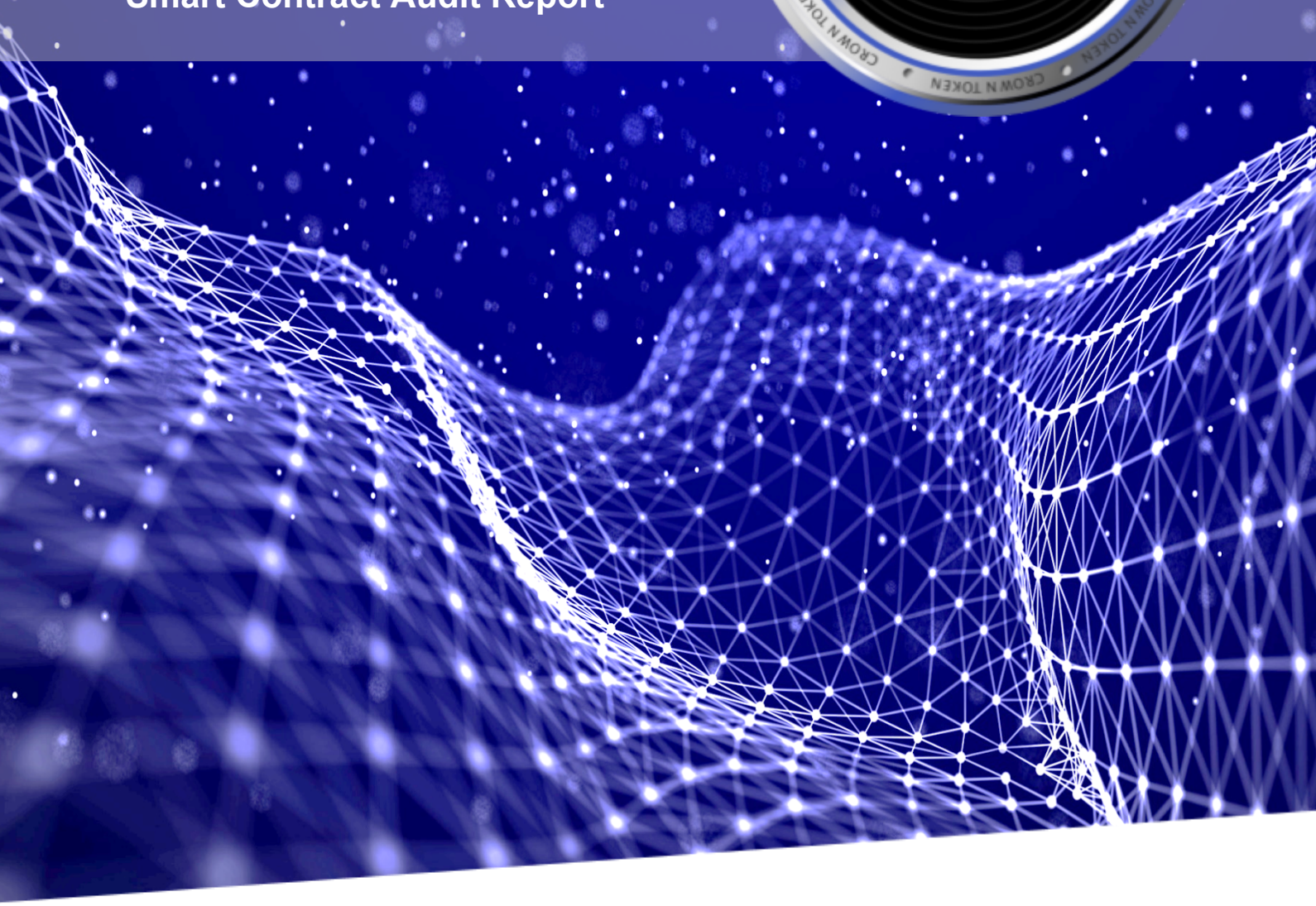
Vega Investment Group Limited

CrownToken and VucaStaking

Smart Contract Audit Report



Date Issued: 2 Dec 2022

Version: Final v1.0

ValiX Consulting

Public

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **CrownToken and VucaStaking features**. This audit report was published on `2 Dec 2022`. The audit scope is limited to the **CrownToken and VucaStaking features.** Our security best practices strongly recommend that the **Vega Investment Group team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About CrownToken and VucaStaking

CROWN token is an entertainment token that bridges traditional IPs with blockchain technology to enhance the core business and create additional value for both IP owners and the community. The token is supported by high-quality IP projects, including animated movies, series, and live-action films. Users can be relevant in the entertainment and IPs industry value chain by utilizing the Staking feature of CROWN token by putting CROWN token in staking smart contracts created by VUCA on *adotmarketplace.com*. The reward for the staking pool derives from many streams, e.g., marketing or community development campaigns.

Scope of Work

The security audit conducted does not replace the full security audit of the overall Vega Investment Group's protocol. The scope is limited to the **CrownToken and VucaStaking features** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *CrownToken smart contract* ▪ *VucaStaking smart contract* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/pellartech/vuca-blockchain-public*
Audit Commit	▪ *13fcd040cac4e00d4a2df2adfbd31aaaffa09ecd (branch: main)*

Certified Commit	▪ *3fafff9d7ddc1c1c868e350e3ec250f2d4e784ae (branch: main)*
Audited Files	▪ *./contracts/CrownToken.sol* ▪ *./contracts/VucaOwnable.sol* ▪ *./contracts/VucaStaking.sol* ▪ *Other imported associated Solidity files*
Excluded Files/Contracts	▪ *./contracts/mock/CWT.sol* ▪ *./contracts/mock/USDT.sol*

Remark: Our security best practices strongly recommend that the Vega Investment Group team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	**Anak Mirasing** **Atitawat Pol-in** **Kritsada Dechawattana** **Parichaya Thanawuthikrai** **Phuwanai Thummavet**
Authors	**Anak Mirasing** **Atitawat Pol-in** **Kritsada Dechawattana** **Parichaya Thanawuthikrai** **Phuwanai Thummavet**
Reviewers	**Sumedt Jitpukdebodin**

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **CrownToken and VucaStaking features** have sufficient security protections to be safe for use.

Initially, Valix was able to identify **36 issues** that were categorized from the "Critical" to "Informational" risk level in the given timeframe of the assessment.

For the reassessment, the *Vega Investment Group* team **fixed all critical issues but left 1 high issue acknowledged** due to their business requirement. Besides, the team **left 2 medium issues acknowledged, 1 low issue partially fixed, 1 low issue acknowledged, and 1 informational issue acknowledged**.

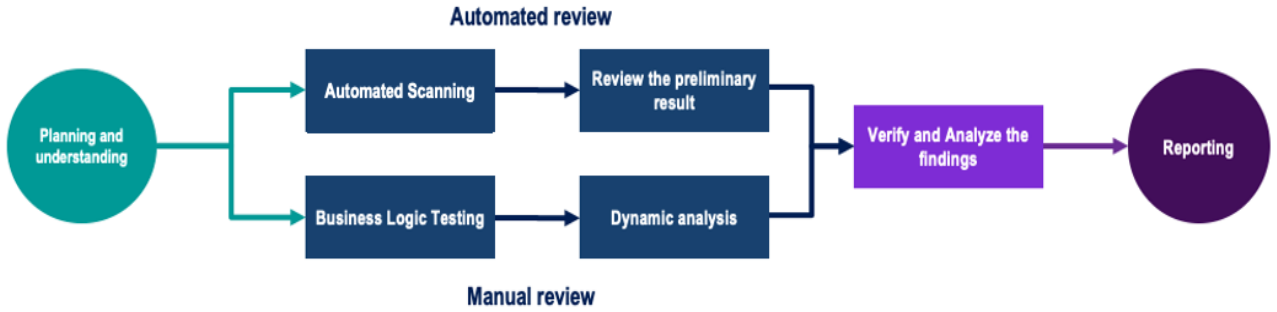
Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
CrownToken and VucaStaking	2	10	10	10	4	0	1	2	2	1

Note: *Risk Rating* **C** *Critical,* **H** *High,* **M** *Medium,* **L** *Low,* **I** *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application's purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	SEC01	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	FNC01	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	OPT01	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	DEV01	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Potentially Draining Pools' Reward Tokens	Critical	Fixed	In use
2	Depending On Incorrect Reward Token Balance #1	Critical	Fixed	In use
3	Potential Denial-Of-Service On Staking Pools	High	Fixed	In use
4	Potential Overriding Pool Changes	High	Fixed	In use
5	Updating Staking End Block Could Lead To State Inconsistency	High	Fixed	In use
6	Incorrectly Calculating Staking Rewards	High	Fixed	In use
7	Potential Denial-Of-Service On Calculating Staker's Rewards	High	Fixed	In use
8	Incorrect Logic Design Of Globally Shared Pool Of Funds	High	Fixed	In use
9	Improperly Sharing Staking Pool's Tokens Balance	High	Acknowledged	In use
10	Incorrectly Sharing Reward Token Balance Between Staking Pools	High	Fixed	In use
11	Improperly Updating Staking Pool Parameters	High	Fixed	In use
12	Incorrectly Applying Pool Changes	High	Fixed	In use
13	Possibly Stealing All Pools' Staking and Reward Tokens	Medium	Fixed	In use
14	Incorrect Calculation Of Withdrawable Pool Rewards #1	Medium	Fixed	In use
15	Depending On Incorrect Reward Token Balance #2	Medium	Fixed	In use
16	Lack Of Guaranteeing Pool State Consistency	Medium	Fixed	In use
17	Usage Of Unsafe Token Transfer Functions	Medium	Fixed	In use
18	Removal Recommendation For Mock Function	Medium	Fixed	In use
19	Possibly Permanent Ownership Removal	Medium	Fixed	In use
20	Unsafe Ownership Transfer	Medium	Fixed	In use

21	Recommended Improving Transparency And Trustworthiness Of Privileged Operations	Medium	Acknowledged	In use
22	Users Can Mistakenly Transfer Reward Tokens To Staking Pools	Medium	Acknowledged	In use
23	Incorrect Calculation Of Withdrawable Pool Rewards #2	Low	Fixed	In use
24	Possibly Unstaking Or Retrieving Reward Tokens Before Staking Period Ends	Low	Fixed	In use
25	Recommended Event Emissions For Transparency And Traceability	Low	Partially Fixed	In use
26	Compiler Is Not Locked To Specific Version	Low	Fixed	In use
27	Compiler May Be Susceptible To Publicly Disclosed Bugs	Low	Fixed	In use
28	Lack Of Applying Pool Changes	Low	Fixed	In use
29	Incorrectly Calculating Total Pool Rewards	Low	Fixed	In use
30	Incorrectly Calculating User's Pool Rewards	Low	Fixed	In use
31	Lack Of Proper Input Sanitization Check	Low	Acknowledged	In use
32	Malfunction Of The depositPoolReward Function	Low	Fixed	In use
33	Inconsistent Error Message With The Code	Informational	Fixed	In use
34	Inconsistent Event Emission With The Code #1	Informational	Fixed	In use
35	Recommended Enforcing Checks-Effects-Interactions Pattern	Informational	Fixed	In use
36	Inconsistent Event Emission With The Code #2	Informational	Acknowledged	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Potentially Draining Pools' Reward Tokens		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 335 - 346*		

Detailed Issue

We detected that the *withdrawERC20* function allows an owner to mistakenly drain all (specific) reward tokens locked in the *VucaStaking* contract, which might be the shared funds from multiple staking pools.

The *withdrawERC20* function could also be adopted by an attacker to steal all reward tokens.

Consider the following two scenarios that can exploit the issue.

1. **Since the *withdrawERC20* function does not record the amount (the *_amount* parameter in L345 in the code snippet below) of the withdrawn reward tokens, the function allows an owner to mistakenly withdraw reward tokens more than the actual amount rewarded to that specific pool.**

   As a result, all reward tokens could be drained from the *VucaStaking* contract.

2. **An attacker with a compromised owner account can drain all reward tokens locked in the contract by adding a new (dummy) short-lived pool and waiting for the end of the pool staking. Next, the attacker can drain all reward tokens by inputting the total balance of the locked rewards into the *withdrawERC20* function.**

VucaStaking.sol

```
335 function withdrawERC20(
336     uint16 _poolId,
337     address _to,
338     uint256 _amount
339 ) external onlyOwner {
340     _updatePoolInfo(_poolId);
341     Pool memory pool = pools[_poolId];
342     require(pool.endBlock <= block.number, "Staking active");
343     require(pool.tokensStaked == 0, "Not allowed");
344 
345     IERC20(pool.rewardToken).transfer(_to, _amount);
346 }
```

Listing 1.1 The *withdrawERC20* function that could drain all reward tokens

The root cause of this issue is that the *withdrawERC20* function **does not account for the amount (the *_amount* parameter in L345) of the withdrawn reward tokens on each staking pool**. Therefore, the function would allow an owner or attacker to withdraw reward tokens multiple times as long as the locked tokens are available.

Recommendations

We recommend updating the *withdrawERC20* function as the code snippet below.

The *withdrawERC20* function would **check for the reward amount available to withdraw (of each specific staking pool) against the input parameter *_amount* (L348)**.

Then, the function would account for the withdrawn amount (L350) before transferring the reward tokens to the specified address, *_to*.

VucaStaking.sol

```
335 function withdrawERC20(
336     uint16 _poolId,
337     address _to,
338     uint256 _amount
339 ) external onlyOwner {
340     _updatePoolInfo(_poolId);
341     Pool storage pool = pools[_poolId];
342     require(pool.endBlock <= block.number, "Staking active");
343     require(pool.tokensStaked == 0, "Not allowed");
344 
345     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
346     uint256 rewardsWithdrew = pool.rewardsWithdrew /
```

```
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
347
348     require(totalUserRewards - rewardsWithdrew >= _amount, "Insufficient pool
    rewards");
349
350     pool.rewardsWithdrew += _amount * (10**IERC20(pool.stakeToken).decimals()) *
    REWARDS_PRECISION;
351
352     IERC20(pool.rewardToken).transfer(_to, _amount);
353 }
```

Listing 1.2 The improved *withdrawERC20* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team decided to remove the *withdrawERC20* function. Hence, this issue was closed.

No. 2	Depending On Incorrect Reward Token Balance #1		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 312 - 332*		

Detailed Issue

We discovered that the *retrieveReward* function depends on the incorrect reward token balance (L326 and L329 in the code snippet below), **leading to potentially draining all (specific) reward tokens locked in the *VucaStaking* contract, which might be the shared funds from multiple staking pools.**

Consider the following two scenarios to exploit the issue.

1. **Since the *retrieveReward* function does not record the amount (the *_amount* parameter in L331) of the withdrawn reward tokens, the function allows an owner to mistakenly withdraw reward tokens more than the actual amount rewarded to that specific pool.**

   As a result, all reward tokens could be drained from the *VucaStaking* contract.

2. **An attacker with a compromised owner account can drain all reward tokens locked in the contract by adding a new (dummy) short-lived pool and waiting for the end of the pool staking. Next, the attacker can drain all reward tokens by inputting the total balance of the locked rewards into the *retrieveReward* function.**

VucaStaking.sol

```
312 function retrieveReward(
313     uint16 _poolId,
314     address _to,
315     uint256 _amount
316 ) external onlyOwner {
317     _updatePoolInfo(_poolId);
318     Pool memory pool = pools[_poolId];
319     require(pool.endBlock <= block.number, "Staking active");
320
321     _updatePoolRewards(_poolId, block.number);
322     pool = pools[_poolId];
```

```
323
324     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
325     uint256 rewardsWithdrew = pool.rewardsWithdrew /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
326     uint256 contractBalance = IERC20(pool.rewardToken).balanceOf(address(this));
327
328     // maximum amount withdrawal = balance - max claimable
329     require(_amount + totalUserRewards <= contractBalance + rewardsWithdrew);
330
331     IERC20(pool.rewardToken).transfer(_to, _amount);
332 }
```

Listing 2.1 The *retrieveReward* function
that depends on the incorrect reward token balance

The root cause of this issue is that **the *retrieveReward* function depends on the incorrect reward token balance (L326) which could represent the total balance aggregated from multiple staking pools.** Hence, the ***require*** statement (L329) that checks for a maximum withdrawable amount would be performed incorrectly.

Furthermore, the *retrieveReward* function also **does not account for the amount (the *_amount* parameter in L331) of the withdrawn reward tokens on each staking pool**. Therefore, the function would allow an owner or attacker to withdraw reward tokens multiple times as long as the locked tokens are available.

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the *retrieveReward* function and related subsystems **to track each pool's *staking* and *reward tokens* isolatedly**.

Reassessment

The *Vega Investment Group* team fixed this issue by reworking the *createPool* function (L191 in the code snippet below) to **allow the creation of only one staking pool for each *VucaStaking* contract**.

VucaStaking.sol

```
179 function createPool(
180     address _rewardToken,
181     address _stakeToken,
182     uint256 _maxStakeTokens,
183     uint256 _startBlock,
184     uint256 _endBlock,
185     uint256 _rewardTokensPerBlock,
```

```
186     uint32 _updateDelay
187 ) external onlyOwner {
188     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
189     require(_rewardToken != address(0), "Invalid reward token");
190     require(_stakeToken != address(0), "Invalid staking token");
191     require(currentPoolId == 0, "Staking pool was already created");
192 
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196 
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
200 
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204 
205     emit PoolCreated(1, currentPoolId, pools[currentPoolId], block.number);
206     currentPoolId += 1;
207 }
```

Listing 2.2 The *createPool* function
that allows the creation of only one staking pool

No. 3	Potential Denial-Of-Service On Staking Pools		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 73 - 78, 81 - 104, 114 - 125, 128 - 150, 153 - 176, 179 - 208, 312 - 332, 335 - 346, and 349 - 371*		

Detailed Issue

We noticed the ***potential denial-of-service issue*** affecting the following functions of the *VucaStaking* contract.

1. ***getRewards* function** (L73 - 78)
2. ***getLatestPoolInfo* function** (L81 - 104)
3. ***getRewardsWithdrawable* function** (L114 - 125)
4. ***stake* function** (L128 - 150)
5. ***emergencyWithdraw* function** (L153 - 176)
6. ***unStake* function** (L179 - 208)
7. ***retrieveReward* function** (L312 - 332)
8. ***withdrawERC20* function** (L335 - 346)
9. ***_updatePoolInfo* function** (L349 - 371)

The root cause of this issue is due to each affected function requiring the process of validating and applying active pool changes (to a specific staking pool) to be done before executing the function's main functionality.

Two functions that are the root cause of the denial-of-service issue include the ***_updatePoolInfo* function** (code snippet 3.1) and the ***getLatestPoolInfo* function** (code snippet 3.2).

The affected functions depending on the ***_updatePoolInfo*** function are as follows.

- ***stake* function** (L128 - 150)
- ***emergencyWithdraw* function** (L153 - 176)
- ***unStake* function** (L179 - 208)
- ***retrieveReward* function** (L312 - 332)
- ***withdrawERC20* function** (L335 - 346)

The affected functions depending on the ***getLatestPoolInfo*** function are as follows.

- ***getRewards* function** (L73 - 78)
- ***getLatestPoolInfo* function** (L81 - 104)
- ***getRewardsWithdrawable* function** (L114 - 125)
- ***unStake* function** (L179 - 208)

Inside the ***_updatePoolInfo*** and ***getLatestPoolInfo*** functions, there are the *for-loops* that iterate through the *poolsChanges* array of each specific staking pool (L352 - 370 in code snippet 3.1 and 84 - 104 in code snippet 3.2). The *loop* would iterate over all array elements to look for active pool changes and apply the changes to the pool.

We found that this process can consume a lot of gas and the gas used is prone to exceeding the block gas limit if the length of the *poolsChanges* array and/or the number of the active pool changes are too large. In the case of exceeding the block gas limit, a transaction would be reverted, leading to the denial-of-service issue to the affected functions.

VucaStaking.sol

```
349 function _updatePoolInfo(uint16 _poolId) internal {
350     Pool storage pool = pools[_poolId];
351 
352     uint256 size = poolsChanges[_poolId].length;
353     for (uint256 i; i < size; i++) {
354         PoolChanges storage changes = poolsChanges[_poolId][i];
355 
356         if (changes.applied) {
357             continue;
358         }
359 
360         uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
361         if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
362             continue;
363         }
364 
365         _updatePoolRewards(_poolId, updateAtBlock);
366         pool.maxStakeTokens = changes.maxStakeTokens;
367         pool.endBlock = changes.endBlock;
368         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
369         changes.applied = true;
370     }
371 }
```

Listing 3.1 The *_updatePoolInfo* function that iterates to apply all active pool changes for a specific staking pool

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
109
110     return pool;
111 }
```

Listing 3.2 The *getLatestPoolInfo* function that iterates over pool changes to simulate the latest info for a specific staking pool

The code snippet 3.3 below presents the *emergencyWithdraw*, one of the affected functions, that would execute the *_updatePoolInfo* function (L154) to apply active pool changes before transferring staking tokens to a staker (function caller).

In case the *_updatePoolInfo* function consumes more gas than the block gas limit, all stakers (including even platform owners) would not be able to interact with the staking pool anymore. This issue also includes the case of stakers withdrawing their staking tokens via the *emergencyWithdraw* function.

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
154     _updatePoolInfo(_poolId);
155     Pool storage pool = pools[_poolId];
156     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
157     uint256 amount = staking.amount;
158     require(staking.amount > 0, "Insufficient funds");
159 
160     _updatePoolRewards(_poolId, block.number);
161     // Update pool
162     if (pool.tokensStaked >= amount) {
163       pool.tokensStaked -= amount;
164     }
165 
166     staking.amount = 0;
167 
168     // Withdraw tokens
169     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
170 
171     emit StakingChanged(msg.sender, _poolId, pool, staking);
172 
173     // Update staker
174     staking.accumulatedRewards = 0;
175     staking.minusRewards = 0;
176 }
```

Listing 3.3 One of the affected functions, *emergencyWithdraw*, executing the *_updatePoolInfo* function

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the pool change update mechanism.

One possible solution is to apply the pagination concept for batch updates of pool changes. Specifically, the large number of pending pool changes would be divided into smaller batch updates. All pending pool changes must be updated sequentially when they are in active blocks only.

This way, the pool change update mechanism can guarantee that there would be no conflict when applying changes and can prevent the update from the denial-of-service issue.

Reassessment

The *Vega Investment Group* team fixed this issue by allowing the maximum number of pending pool changes in the queue (for each staking pool) to be 10.

No. 4	Potential Overriding Pool Changes		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 81 - 111, 267 - 275, 277 - 287, 290 - 299, and 349 - 371*		

Detailed Issue

Code snippet 4.1 presents the functions ***updateMaxStakeTokens*** (L267 - 275), ***updateRewardTokensPerBlock*** (L277 - 287), and ***updateEndBlock*** (L290 - 299), which allow an owner to update pool parameters (i.e., ***maxStakeTokens***, ***rewardTokensPerBlock***, and ***endBlock*** respectively) of a particular pool.

Once an owner triggers one of those functions, a pending *pool change* order would be created and it would be applied by the functions ***_updatePoolInfo*** (L366 - 368 in code snippet 4.2) and ***getLatestPoolInfo*** (L101 - 103 in code snippet 4.3) at its (future) active block number.

We discovered that, with this pool change update mechanism, some pending *pool changes* could potentially be conflicted after they are applied to the pool.

Consider the following pool change update scenario to understand the issue.

- **PoolChange #1:** For updating the ***maxStakeTokens*** **parameter to 100** was created and would be **active at block number 3000**.

  PoolChange: {
  maxStakeTokens: ***100*** (the parameter that an owner wanted to update),
  endBlock: ***5000*** (loaded from the contract storage),
  rewardTokensPerBlock: ***50*** (loaded from the contract storage),
  activeBlock: ***3000***
  }

- **PoolChange #2:** For updating the ***rewardTokensPerBlock*** **parameter to 200** was created and would be **active at block number 3001**.

  PoolChange: {
  maxStakeTokens: ***70*** (loaded from the contract storage;
  100 in the PoolChange #1 was not yet applied),
  endBlock: ***5000*** (loaded from the contract storage),
  rewardTokensPerBlock: ***200*** (the parameter that an owner wanted to update),
  activeBlock: ***3001***
  }

Immediately after both *pool changes* have been applied to the pool, the parameter *maxStakeTokens* would store *70* (which is the old value loaded from the contract storage at the time creating the *PoolChange #2*; the new value of *100* in the *PoolChange #1* would not be effective on the pool as expected).

This issue could lead to *incorrect pool parameter configurations*. And, the owner has no way of knowing which *pool changes* have been committed but not been applied to the pool.

The root cause of this issue is that the structure of the *pool change* payload contains all three pool parameters ***rewardTokensPerBlock***, ***endBlock***, and ***maxStakeTokens*** (L271 in code snippet 4.1). But, when each *pool change* order is created, only a single parameter would be required to get updated at a time and the other parameters would be loaded from the contract storage. Hence, this incorrect update mechanism could lead to the *pool change overriding* issue.

VucaStaking.sol

```
267 function updateMaxStakeTokens(uint16 _poolId, uint256 _maxStakeTokens) external
    onlyOwner {
268     require(pools[_poolId].inited, "Invalid Pool");
269     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
270
271     PoolChanges memory changes = PoolChanges({ applied: false,
    rewardTokensPerBlock: pools[_poolId].rewardTokensPerBlock, endBlock:
    pools[_poolId].endBlock, maxStakeTokens: _maxStakeTokens, timestamp:
    block.timestamp, blockNumber: block.number });
272     poolsChanges[_poolId].push(changes);
273
274     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
275 }
276
277 function updateRewardTokensPerBlock(uint16 _poolId, uint256
    _rewardTokensPerBlock) external onlyOwner {
278     require(pools[_poolId].inited, "Invalid Pool");
```

```
279     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
280
281     uint256 rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(pools[_poolId].stakeToken).decimals()) * REWARDS_PRECISION;
282
283     PoolChanges memory changes = PoolChanges({ applied: false,
    rewardTokensPerBlock: rewardTokensPerBlock, endBlock: pools[_poolId].endBlock,
    maxStakeTokens: pools[_poolId].maxStakeTokens, timestamp: block.timestamp,
    blockNumber: block.number });
284     poolsChanges[_poolId].push(changes);
285
286     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
287 }
288
289 // end block updatable
290 function updateEndBlock(uint16 _poolId, uint256 _endBlock) external onlyOwner {
291     require(pools[_poolId].inited, "Invalid Pool");
292     require(block.number <= _endBlock, "Invalid input");
293     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
294
295     PoolChanges memory changes = PoolChanges({ applied: false,
    rewardTokensPerBlock: pools[_poolId].rewardTokensPerBlock, endBlock: _endBlock,
    maxStakeTokens: pools[_poolId].maxStakeTokens, timestamp: block.timestamp,
    blockNumber: block.number });
296     poolsChanges[_poolId].push(changes);
297
298     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
299 }
```

Listing 4.1 The *updateMaxStakeTokens*, *updateRewardTokensPerBlock*, and *updateEndBlock* functions

VucaStaking.sol

```
349 function _updatePoolInfo(uint16 _poolId) internal {
350     Pool storage pool = pools[_poolId];
351
352     uint256 size = poolsChanges[_poolId].length;
353     for (uint256 i; i < size; i++) {
354         PoolChanges storage changes = poolsChanges[_poolId][i];
355
356         if (changes.applied) {
357             continue;
358         }
359
```

```
360         uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
361         if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
362             continue;
363         }
364
365         _updatePoolRewards(_poolId, updateAtBlock);
366         pool.maxStakeTokens = changes.maxStakeTokens;
367         pool.endBlock = changes.endBlock;
368         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
369         changes.applied = true;
370     }
371 }
```

Listing 4.2 The *_updatePoolInfo* function that applies pool changes to the pool

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
109
110     return pool;
```

```
111 }
```

Listing 4.3 The *getLatestPoolInfo* function that employs pool changes

Recommendations

We recommend revising all the associated functions and data structures. In code snippet 4.4 below, we revised the functions ***updateMaxStakeTokens*** (L267 - 275), ***updateRewardTokensPerBlock*** (L277 - 287), and ***updateEndBlock*** (L290 - 299) to support updating only a single pool parameter at a time.

We also improved the functions ***_updatePoolInfo*** (L367 - 375 in code snippet 4.5) and ***getLatestPoolInfo*** (L101 - 109 in code snippet 4.6) to update only a single pool parameter over each *pool change* order.

VucaStaking.sol

```
267 function updateMaxStakeTokens(uint16 _poolId, uint256 _maxStakeTokens) external
    onlyOwner {
268     require(pools[_poolId].inited, "Invalid Pool");
269     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
270
271     PoolChanges memory changes = PoolChanges({ applied: false, updateParam:
    UpdateParam.MaxStakeTokens, updateParamValue: _maxStakeTokens, timestamp:
    block.timestamp, blockNumber: block.number });
272     poolsChanges[_poolId].push(changes);
273
274     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
275 }
276
277 function updateRewardTokensPerBlock(uint16 _poolId, uint256
    _rewardTokensPerBlock) external onlyOwner {
278     require(pools[_poolId].inited, "Invalid Pool");
279     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
280
281     uint256 rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(pools[_poolId].stakeToken).decimals()) * REWARDS_PRECISION;
282
283     PoolChanges memory changes = PoolChanges({ applied: false, updateParam:
    UpdateParam.RewardTokensPerBlock, updateParamValue: rewardTokensPerBlock,
    timestamp: block.timestamp, blockNumber: block.number });
284     poolsChanges[_poolId].push(changes);
285
286     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
287 }
288
```

```
289 // end block updatable
290 function updateEndBlock(uint16 _poolId, uint256 _endBlock) external onlyOwner {
291     require(pools[_poolId].inited, "Invalid Pool");
292     require(block.number <= _endBlock, "Invalid input");
293     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
294
295     PoolChanges memory changes = PoolChanges({ applied: false, updateParam:
    UpdateParam.EndBlock, updateParamValue: _endBlock, timestamp: block.timestamp,
    blockNumber: block.number });
296     poolsChanges[_poolId].push(changes);
297
298     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
299 }
```

Listing 4.4 The improved *updateMaxStakeTokens*, *updateRewardTokensPerBlock*, and *updateEndBlock* functions that support updating only a single pool parameter at a time

VucaStaking.sol

```
349 function _updatePoolInfo(uint16 _poolId) internal {
350     Pool storage pool = pools[_poolId];
351
352     uint256 size = poolsChanges[_poolId].length;
353     for (uint256 i; i < size; i++) {
354         PoolChanges storage changes = poolsChanges[_poolId][i];
355
356         if (changes.applied) {
357             continue;
358         }
359
360         uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
361         if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
362             continue;
363         }
364
365         _updatePoolRewards(_poolId, updateAtBlock);
366
367         if (changes.updateParam == UpdateParam.MaxStakeTokens) {
368             pool.maxStakeTokens = changes.updateParamValue;
369         }
370         else if (changes.updateParam == UpdateParam.EndBlock) {
371             pool.endBlock = changes.updateParamValue;
372         }
373         else if (changes.updateParam == UpdateParam.RewardTokensPerBlock) {
374             pool.rewardTokensPerBlock = changes.updateParamValue;
375         }
376
```

```
377             changes.applied = true;
378         }
379     }
```

Listing 4.5 The improved *_updatePoolInfo* function

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         if (changes.updateParam == UpdateParam.MaxStakeTokens) {
102             pool.maxStakeTokens = changes.updateParamValue;
103         }
104         else if (changes.updateParam == UpdateParam.EndBlock) {
105             pool.endBlock = changes.updateParamValue;
106         }
107         else if (changes.updateParam == UpdateParam.RewardTokensPerBlock) {
108             pool.rewardTokensPerBlock = changes.updateParamValue;
109         }
110     }
111
112     uint256 _rewards;
113     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
114     pool.totalUserRewards += _rewards;
115
116     return pool;
117 }
```

Listing 4.6 The improved *getLatestPoolInfo* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team adopted our recommended code to fix this issue.

No. 5	Updating Staking End Block Could Lead To State Inconsistency		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 81 - 111, 290 - 299, and 349 - 371*		

Detailed Issue

The *updateEndBlock* function (code snippet 5.1) allows an owner to create **a pool change order for updating the *endBlock* parameter** (L295 - 296) for a specified pool id. The pool change order would be applied to the pool by the following functions: *_updatePoolInfo* (L367 in code snippet 5.2) and *getLatestPoolInfo* (L102 in code snippet 5.3).

However, when applying the ***endBlock*** parameter to the pool, we detected the conflict possibility, leading to the state inconsistency issue.

More specifically, if the new ***endBlock*** parameter (L295 in code snippet 5.1) is less than or equal to its active block number (*block.number* + *pools[_poolId].updateDelay*). The conflict would occur if other pool changes with an active block number more than the new ***endBlock*** parameter were applied before the new ***endBlock*** parameter is effective.

In other words, all pending pool changes with an active block number more than the new ***endBlock*** parameter would become invalid, and they would cause state inconsistency suddenly after they are applied to the pool.

To understand this issue better, consider the following pool change update scenario.

- **PoolChange #1:** For updating the ***rewardTokensPerBlock*** **parameter to 100** was created and would be **active at block number 1000**.
- **PoolChange #2:** For updating the ***maxStakeTokens*** **parameter to 5000** was created and would be **active at block number 1001**.
- **PoolChange #3:** For updating the ***endBlock*** **parameter to 900** was created and would be **active at block number 1002**.

Suddenly after all three pool changes above have been applied, the pool would end staking at block number 900 as per *PoolChange #3*, making the *PoolChange #1* and *PoolChange #2* that had been applied previously became invalid, rendering the inconsistent state to that pool.

The root cause of the issue is that the *updateEndBlock* function lacks proper validation on the ***endBlock*** parameter (L292 in code snippet 5.1).

VucaStaking.sol

```
290 function updateEndBlock(uint16 _poolId, uint256 _endBlock) external onlyOwner {
291     require(pools[_poolId].inited, "Invalid Pool");
292     require(block.number <= _endBlock, "Invalid input");
293     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
294
295     PoolChanges memory changes = PoolChanges({ applied: false,
    rewardTokensPerBlock: pools[_poolId].rewardTokensPerBlock, endBlock: _endBlock,
    maxStakeTokens: pools[_poolId].maxStakeTokens, timestamp: block.timestamp,
    blockNumber: block.number });
296     poolsChanges[_poolId].push(changes);
297
298     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
299 }
```

Listing 5.1 The *updateEndBlock* function that allows an owner to create a pool change order for updating the *endBlock* parameter for a specified pool id

VucaStaking.sol

```
349 function _updatePoolInfo(uint16 _poolId) internal {
350     Pool storage pool = pools[_poolId];
351
352     uint256 size = poolsChanges[_poolId].length;
353     for (uint256 i; i < size; i++) {
354         PoolChanges storage changes = poolsChanges[_poolId][i];
355
356         if (changes.applied) {
357             continue;
358         }
359
360         uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
361         if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
362             continue;
363         }
364
365         _updatePoolRewards(_poolId, updateAtBlock);
366         pool.maxStakeTokens = changes.maxStakeTokens;
367         pool.endBlock = changes.endBlock;
```

```
368         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
369         changes.applied = true;
370     }
371 }
```

Listing 5.2 The *_updatePoolInfo* function that applies the new *endBlock* parameter for a specified pool id

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
109
110     return pool;
111 }
```

Listing 5.3 The *getLatestPoolInfo* function that applies the new *endBlock* parameter to simulate the latest info for a specific staking pool

Recommendations

As discussed earlier, the root cause of this issue is that the *updateEndBlock* function lacks proper validation on the ***endBlock*** parameter. We recommend validating the ***endBlock*** parameter with the following *require* statement (L292 in the code snippet below).

require(_endBlock > block.number + pools[_poolId].updateDelay, "Invalid input");

This validation check would guarantee that the new ***endBlock*** parameter must always be more than its active block number, preventing the state inconsistency issue.

VucaStaking.sol

```
290 function updateEndBlock(uint16 _poolId, uint256 _endBlock) external onlyOwner {
291     require(pools[_poolId].inited, "Invalid Pool");
292     require(_endBlock > block.number + pools[_poolId].updateDelay, "Invalid
    input");
293     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
294
295     PoolChanges memory changes = PoolChanges({ applied: false,
    rewardTokensPerBlock: pools[_poolId].rewardTokensPerBlock, endBlock: _endBlock,
    maxStakeTokens: pools[_poolId].maxStakeTokens, timestamp: block.timestamp,
    blockNumber: block.number });
296     poolsChanges[_poolId].push(changes);
297
298     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number +
    pools[_poolId].updateDelay);
299 }
```

Listing 5.4 The improved *updateEndBlock* function
with proper validation on the *endBlock* parameter

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue as per our suggestion.

No. 6	Incorrectly Calculating Staking Rewards		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 211 - 233 and 236 - 265*		

Detailed Issue

We discovered an incorrect input validation on the *_startBlock* parameter on the *createPool* function (L245 in code snippet 6.1) leading to incorrectly calculating staking rewards.

If the *_startBlock* parameter was inputted less than the current block number (*block.number*), the incorrect *_startBlock*'s value would become the parameter *pool.lastRewardedBlock* (L258 in code snippet 6.1) in the following formula (L226 in code snippet 6.2).

blocksSinceLastReward = floorBlock - pool.lastRewardedBlock

The computed *blocksSinceLastReward* would be more than the expected value which would cause the calculated *rewards* (L228 in code snippet 6.2) for stakers of that pool to be more than the actual amount. Subsequently, **the platform owner would have to pay stakers more reward tokens than expected.**

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248 
249         pools[currentPoolId].inited = true;
250         pools[currentPoolId].rewardToken = _rewardToken;
```

```
251         pools[currentPoolId].stakeToken = _stakeToken;
252
253         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
254         pools[currentPoolId].startBlock = _startBlock;
255         pools[currentPoolId].endBlock = _endBlock;
256
257         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
258         pools[currentPoolId].lastRewardedBlock = _startBlock;
259         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
260
261         PoolChanges memory changes;
262
263         emit PoolUpdated(currentPoolId, pools[currentPoolId], changes,
    block.number);
264         currentPoolId += 1;
265 }
```

Listing 6.1 The *createPool* function with incorrect validation on the *_startBlock* parameter

VucaStaking.sol

```
211 function getPoolRewardsCheckpoint(uint16 _poolId, uint256 _blockNumber)
212     public
213     view
214     returns (
215         uint256 accumulatedRewardsPerShare,
216         uint256 lastRewardedBlock,
217         uint256 rewards
218     )
219 {
220     Pool memory pool = pools[_poolId];
221
222     uint256 floorBlock = _blockNumber <= pool.endBlock ? _blockNumber :
    pool.endBlock;
223
224     uint256 blocksSinceLastReward;
225     if (floorBlock >= pool.lastRewardedBlock) {
226         blocksSinceLastReward = floorBlock - pool.lastRewardedBlock;
227     }
228     rewards = blocksSinceLastReward * pool.rewardTokensPerBlock;
229     if (pool.tokensStaked > 0) {
230         accumulatedRewardsPerShare = pool.accumulatedRewardsPerShare + (rewards
    / pool.tokensStaked);
231     }
232     lastRewardedBlock = floorBlock;
233 }
```

Listing 6.2 The *getPoolRewardsCheckpoint* function that could compute incorrect staking rewards

Recommendations

We recommend improving the input validation on the *_startBlock* parameter by checking that **the inputted value must be more than the current block number (*block.number*)** like L245 in the code snippet below.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248 
249         pools[currentPoolId].inited = true;
250         pools[currentPoolId].rewardToken = _rewardToken;
251         pools[currentPoolId].stakeToken = _stakeToken;
252 
253         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
254         pools[currentPoolId].startBlock = _startBlock;
255         pools[currentPoolId].endBlock = _endBlock;
256 
257         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
258         pools[currentPoolId].lastRewardedBlock = _startBlock;
259         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
260 
261         PoolChanges memory changes;
262 
263         emit PoolUpdated(currentPoolId, pools[currentPoolId], changes,
    block.number);
264         currentPoolId += 1;
265 }
```

Listing 6.3 The improved *createPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed as per our recommendation.

No. 7	Potential Denial-Of-Service On Calculating Staker's Rewards		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 73 - 78, 81 - 111, and 179 - 208*		

Detailed Issue

The *unStake* function (code snippet 7.1) allows stakers to withdraw their staking tokens and retrieve their reward tokens after the staking period ends. To calculate the rewards for each staker, the *unStake* function executes the *getRewards* function (L190).

Then, the *getRewards* function invokes the *getLatestPoolInfo* function to get the up-to-date pool's parameters (L75 in code snippet 7.2). The *getLatestPoolInfo* function has to iterate over all pool changes (contained in the *poolsChanges* array) to simulate the up-to-date pool's parameters (L84 - 104 in code snippet 7.3).

We noticed that the process of simulating the up-to-date pool's parameters in the *getLatestPoolInfo* function can consume more gas than the block gas limit if the number of pool changes is too large, causing the unstaking transaction to be reverted.

Moreover, we also noticed that the *getRewards* function actually consumes only the static pool parameter ***pool.stakeToken*** (L77 in code snippet 7.2) which could be directly loaded from the contract storage.

For this reason, we consider that invoking the *getLatestPoolInfo* function (by the *getRewards* function) without utilizing any dynamic pool parameters would highly increase the opportunity for the unstaking transaction to be reverted due to exceeding the block gas limit.

VucaStaking.sol

```
179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock <= block.number, "Staking active");
183 
184     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
185     uint256 amount = staking.amount;
```

```
186     require(staking.amount > 0, "Insufficient funds");
187
188     _updatePoolRewards(_poolId, block.number);
189     // Pay rewards
190     uint256 rewards = getRewards(_poolId, msg.sender);
191     IERC20(pool.rewardToken).transfer(msg.sender, rewards);
192
193     // Update pool
194     pool.rewardsWithdrew += getRawRewards(_poolId, msg.sender);
195     if (pool.tokensStaked >= amount) {
196       pool.tokensStaked -= amount;
197     }
198
199     // Withdraw tokens
200     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
201
202     emit StakingChanged(msg.sender, _poolId, pool, staking);
203
204     // Update staker
205     staking.accumulatedRewards = 0;
206     staking.minusRewards = 0;
207     staking.amount = 0;
208 }
```

Listing 7.1 The *unStake* function executes the *getRewards* function to calculate staker's rewards for a specific staking pool

VucaStaking.sol

```
73  function getRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
74      uint256 rawRewards = getRawRewards(_poolId, _account);
75      Pool memory pool = getLatestPoolInfo(_poolId);
76
77      return rawRewards / (10**IERC20(pool.stakeToken).decimals()) /
    REWARDS_PRECISION;
78  }
```

Listing 7.2 The *getRewards* function invokes the *getLatestPoolInfo* function to get the up-to-date parameters for a specific staking pool

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
```

```
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
109
110     return pool;
111 }
```

Listing 7.3 The *getLatestPoolInfo* function that iterates over pool changes to simulate the up-to-date parameters for a specific staking pool, which can consume more gas than the block gas limit

Recommendations

We consider that invoking the *getLatestPoolInfo* function (by the *getRewards* function) without utilizing any dynamic pool parameters would highly increase the opportunity for the unstaking transaction to be reverted due to exceeding the block gas limit.

To mitigate the denial-of-service issue on unstaking transactions, we recommend updating the *getRewards* function by **directly loading the pool's static parameters from the contract storage instead** like L75 in the below code snippet.

VucaStaking.sol

```
73  function getRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
74      uint256 rawRewards = getRawRewards(_poolId, _account);
75      Pool memory pool = pools[_poolId];
76
77      return rawRewards / (10**IERC20(pool.stakeToken).decimals()) /
    REWARDS_PRECISION;
78  }
```

Listing 7.4 The improved *getRewards* function that directly loads the pool's static parameters from the contract storage

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue according to our suggestion.

No. 8	Incorrect Logic Design Of Globally Shared Pool Of Funds		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 153 - 176, 179 - 197, 312 - 332, and 335 - 346*		

Detailed Issue

This issue affects the following functions.

1. ***emergencyWithdraw* function** (**L153 - 176** in *VucaStaking.sol*)
2. ***unStake* function** (**L179 - 197** in *VucaStaking.sol*)
3. ***retrieveReward* function** (**L312 - 332** in *VucaStaking.sol*)
4. ***withdrawERC20* function** (**L335 - 346** in *VucaStaking.sol*)

In the *VucaStaking* contract, multiple staking pools can be created and active simultaneously. **We found that all staking pools that are utilizing the same staking and/or reward token(s) would share their funds as a global single pool.**

Consequently, the invocation of any of the above-listed functions on one staking pool could affect the balance of the other associated staking pools.

For this reason, we considered that the *globally shared pool of funds* was designed incorrectly and the design is prone to several pool imbalance issues.

Imagine if one staking pool's balance is managed incorrectly somehow, that would affect the balance of other staking pools immediately.

Recommendations

We recommend redesigning and reimplementing the associated functions and their subsystems **to separate each staking pool's balance apart**.

Reassessment

The *Vega Investment Group* team fixed this issue by reworking the *createPool* function (L191 in the code snippet below) to **allow the creation of only one staking pool for each *VucaStaking* contract**.

VucaStaking.sol

```
179 function createPool(
180     address _rewardToken,
181     address _stakeToken,
182     uint256 _maxStakeTokens,
183     uint256 _startBlock,
184     uint256 _endBlock,
185     uint256 _rewardTokensPerBlock,
186     uint32 _updateDelay
187 ) external onlyOwner {
188     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
189     require(_rewardToken != address(0), "Invalid reward token");
190     require(_stakeToken != address(0), "Invalid staking token");
191     require(currentPoolId == 0, "Staking pool was already created");
192
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
200
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204
205     emit PoolCreated(1, currentPoolId, pools[currentPoolId], block.number);
206     currentPoolId += 1;
207 }
```

Listing 8.1 The *createPool* function
that allows the creation of only one staking pool

No. 9	Improperly Sharing Staking Pool's Tokens Balance		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 236 - 265*		

Detailed Issue

In the *VucaStaking* contract, the *createPool* function (code snippet below) permits an owner to create a staking pool with the same staking and reward tokens. At this point, **we found the improperly shared tokens balance issues if a pool has the same staking and reward tokens.**

Consider the following two scenarios.

1. **If an owner manages the reward token's balance incorrectly, not every staker would be able to successfully execute the *unStake* function** because the total amount aggregated from the staking and reward tokens of that pool would be insufficient for all stakers.
2. **An owner can withdraw all tokens from the pool via the *retrieveReward* function.**

The root cause of this issue is that the staking token's amount could be viewed as the available tokens for retrieving as the rewards.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248
```

```
249         pools[currentPoolId].inited = true;
250         pools[currentPoolId].rewardToken = _rewardToken;
251         pools[currentPoolId].stakeToken = _stakeToken;
252
253         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
254         pools[currentPoolId].startBlock = _startBlock;
255         pools[currentPoolId].endBlock = _endBlock;
256
257         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
258         pools[currentPoolId].lastRewardedBlock = _startBlock;
259         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
260
261         PoolChanges memory changes;
262
263         emit PoolUpdated(currentPoolId, pools[currentPoolId], changes,
    block.number);
264         currentPoolId += 1;
265 }
```

Listing 9.1 The *createPool* function that permits to create a staking pool with the same staking and reward tokens

Recommendations

We recommend updating the *createPool* function by adding a sanitization check like L248 in the code snippet below **to ensure that each staking pool cannot have the same staking and reward tokens**.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248         require(_stakeToken != _rewardToken, "Staking token cannot be the same
    as the reward token");
249
250         pools[currentPoolId].inited = true;
251         pools[currentPoolId].rewardToken = _rewardToken;
```

```
252         pools[currentPoolId].stakeToken = _stakeToken;
253
254         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
255         pools[currentPoolId].startBlock = _startBlock;
256         pools[currentPoolId].endBlock = _endBlock;
257
258         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
259         pools[currentPoolId].lastRewardedBlock = _startBlock;
260         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
261
262         PoolChanges memory changes;
263
264         emit PoolUpdated(currentPoolId, pools[currentPoolId], changes,
    block.number);
265         currentPoolId += 1;
266 }
```

Listing 9.2 The improved *createPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team acknowledged this issue. **Since using the same staking and reward tokens in a certain pool is a business requirement, the team decided to maintain the original code.**

No. 10	Incorrectly Sharing Reward Token Balance Between Staking Pools		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 150 - 177 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

The *unStake* function (code snippet 10.1) does not check and update the parameter ***pool.extension.totalPoolRewards***.

Hence, the function can view the balance of other staking pools that utilize the same reward token as their staking and/or reward tokens as the reward balance of one particular pool.

As a result, an invocation of the *unStake* function on one staking pool can affect other pools' balance.

Let's consider the following exploitable scenario to understand this issue.

1. Pool-A is created using CROWN as a staking token and USDT as a reward token.
2. Pool-B is created using USDT as a staking token and CROWN as a reward token.
3. User-A stakes 50 CROWN to Pool-A.
4. User-B stakes 50 USDT to Pool-B.
5. Owner deposits 200 CROWN as a pool reward for Pool-B.
6. Both pools reach their staking period.
7. User-A unstakes his 50 CROWN from Pool-A and retrieves 10 USDT as a staking reward.

   In this step, 10 USDT staked by User-B for Pool-B (Step 4) has been withdrawn incorrectly. The *unstake* function did not check and update the *totalPoolRewards parameter*.

8. User-B unstakes his 50 USDT from Pool-B to get 100 CROWN as a staking reward but the transaction is reverted.

   Since the total balance of USDT in the *VucaStaking* contract is now 40, not 50 (10 USDT was withdrawn by User-A in Step 7), User-B cannot unstake his 50 USDT tokens even if an owner had already deposited sufficient CROWN tokens as a reward in Step 5.

Please note that the scenario described above is one of several exploitable scenarios.

VucaStaking.sol

```
150 function unStake(uint16 _poolId) external {
151     _updatePoolInfo(_poolId);
152     Pool storage pool = pools[_poolId];
153     require(pool.endBlock < block.number, "Staking active");
154 
155     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
156     uint256 amount = staking.amount;
157     require(staking.amount > 0, "Insufficient funds");
158 
159     _updatePoolRewards(_poolId, block.number);
160     uint256 rewards = getRewards(_poolId, msg.sender);
161 
162     // Update pool
163     pool.tokensStaked -= amount;
164 
165     emit StakingChanged("StakingChanged", msg.sender, _poolId, pool, staking);
166 
167     // Update staker
168     staking.accumulatedRewards = 0;
169     staking.minusRewards = 0;
170     staking.amount = 0;
171 
172     // Pay rewards
173     IERC20(pool.rewardToken).safeTransfer(msg.sender, rewards);
174 
175     // Withdraw tokens
176     IERC20(pool.stakeToken).safeTransfer(address(msg.sender), amount);
177 }
```

Listing 10.1 The *unStake* function that does not check and update the parameter *pool.extension.totalPoolRewards*

Recommendations

We recommend reworking the *unStake* function to check and update the parameter ***pool.extension.totalPoolRewards*** accordingly and making sure that each staking pool's balance is separated apart.

Please be sure to perform the proper unit testing on all possible edge cases to ensure that all related functions will correctly perform as per your staking model.

Reassessment

The *Vega Investment Group* team fixed this issue by reworking the *createPool* function (L191 in the code snippet below) to **allow the creation of only one staking pool for each *VucaStaking* contract**.

VucaStaking.sol

```
179 function createPool(
180     address _rewardToken,
181     address _stakeToken,
182     uint256 _maxStakeTokens,
183     uint256 _startBlock,
184     uint256 _endBlock,
185     uint256 _rewardTokensPerBlock,
186     uint32 _updateDelay
187 ) external onlyOwner {
188     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
189     require(_rewardToken != address(0), "Invalid reward token");
190     require(_stakeToken != address(0), "Invalid staking token");
191     require(currentPoolId == 0, "Staking pool was already created");
192
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
200
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204
205     emit PoolCreated(1, currentPoolId, pools[currentPoolId], block.number);
206     currentPoolId += 1;
207 }
```

Listing 10.2 The *createPool* function
that allows the creation of only one staking pool

No. 11	Improperly Updating Staking Pool Parameters		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 150 - 177 and 294 - 317 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We discovered that the *unStake* and *retrieveReward* functions do not properly update important parameters.

For the *unStake* function (code snippet 11.1), we found that the function does not update the parameters ***pool.rewardsWithdrew*** and ***pool.extension.totalPoolRewards***.

For the *retrieveReward* function (code snippet 11.2), we found that the function does not update the parameters ***pool.extension.noAddressRewards*** and ***pool.extension.totalPoolRewards***.

Without updating these parameters properly, staking pools could not separate their reward balance.

Please refer to Issues #2 (Depending On Incorrect Reward Token Balance #1), #8 (Incorrect Logic Design Of Globally Shared Pool Of Funds), and #13 (Possibly Stealing All Pools' Staking and Reward Tokens) for more details.

VucaStaking.sol

```
150 function unStake(uint16 _poolId) external {
151     _updatePoolInfo(_poolId);
152     Pool storage pool = pools[_poolId];
153     require(pool.endBlock < block.number, "Staking active");
154 
155     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
156     uint256 amount = staking.amount;
157     require(staking.amount > 0, "Insufficient funds");
158 
159     _updatePoolRewards(_poolId, block.number);
160     uint256 rewards = getRewards(_poolId, msg.sender);
161 
```

```
162     // Update pool
163     pool.tokensStaked -= amount;
164 
165     emit StakingChanged("StakingChanged", msg.sender, _poolId, pool, staking);
166 
167     // Update staker
168     staking.accumulatedRewards = 0;
169     staking.minusRewards = 0;
170     staking.amount = 0;
171 
172     // Pay rewards
173     IERC20(pool.rewardToken).safeTransfer(msg.sender, rewards);
174 
175     // Withdraw tokens
176     IERC20(pool.stakeToken).safeTransfer(address(msg.sender), amount);
177 }
```

Listing 11.1 The *unStake* function

VucaStaking.sol

```
294 function retrieveReward(uint16 _poolId, address _to) external onlyOwner {
295     _updatePoolInfo(_poolId);
296     Pool storage pool = pools[_poolId];
297     require(pool.endBlock < block.number, "Staking active");
298 
299     _updatePoolRewards(_poolId, block.number);
300     pool = pools[_poolId];
301 
302     uint256 totalPoolRewards = pool.extension.totalPoolRewards;
303     uint256 noAddressRewards = pool.extension.noAddressRewards;
304     uint256 rewardsWithdrew = pool.extension.rewardsWithdrew;
305 
306     uint256 totalUserRewards = pool.extension.totalUserRewards /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
307 
308     require(totalPoolRewards + noAddressRewards > totalUserRewards +
    rewardsWithdrew, "Insufficient pool rewards");
309 
310     uint256 amount = totalPoolRewards + noAddressRewards - totalUserRewards -
    rewardsWithdrew;
311 
312     pool.extension.rewardsWithdrew += amount;
313 
314     emit RewardsRetrieved('RewardsRetrieved', _poolId, amount);
315 
316     IERC20(pool.rewardToken).safeTransfer(_to, amount);
317 }
```

Listing 11.2 The *retrieveReward* function

Recommendations

We recommend updating all associated parameters (and also all related functions) by making sure that each staking pool's balance would be isolated from others.

Moreover, we do not assure that the formulas in L308 and L310 in code snippet 11.2 above would function correctly after the associated parameters are updated. Thus, please double-check these formulas thoroughly.

Please be sure to perform the proper unit testing on all possible edge cases to ensure that all related functions would correctly perform as per your staking model.

Reassessment

The *Vega Investment Group* team fixed this issue by reworking the *createPool* function (L191 in the code snippet below) to **allow the creation of only one staking pool for each *VucaStaking* contract**.

VucaStaking.sol

```
179 function createPool(
180     address _rewardToken,
181     address _stakeToken,
182     uint256 _maxStakeTokens,
183     uint256 _startBlock,
184     uint256 _endBlock,
185     uint256 _rewardTokensPerBlock,
186     uint32 _updateDelay
187 ) external onlyOwner {
188     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
189     require(_rewardToken != address(0), "Invalid reward token");
190     require(_stakeToken != address(0), "Invalid staking token");
191     require(currentPoolId == 0, "Staking pool was already created");
192
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
200
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204
205     emit PoolCreated(1, currentPoolId, pools[currentPoolId], block.number);
```

```
206         currentPoolId += 1;
207     }
```

Listing 11.3 The *createPool* function
that allows the creation of only one staking pool

No. 12	Incorrectly Applying Pool Changes		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 229 - 247, 249 - 269, and 272 - 291 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

The following functions execute the *_updatePoolInfo* function incorrectly.

- ***updateMaxStakeTokens*** function (*L229 - 247* in *VucaStaking.sol*)
- ***updateRewardTokensPerBlock*** function (*L249 - 269* in *VucaStaking.sol*)
- ***updateEndBlock*** function (*L272 - 291* in *VucaStaking.sol*)

In code snippet 12.1, the *updateMaxStakeTokens* function executes the *_updatePoolInfo* function in L233 after checking the active block (the ***require*** statement in L231).

We detected that the ***endBlock*** parameter could be updated during the invocation of the *_updatePoolInfo* function in L233. If so, the ***require*** statement in L231 would not process the updated ***endBlock*** parameter, leading to the state inconsistency issue.

VucaStaking.sol

```
229 function updateMaxStakeTokens(uint16 _poolId, uint256 _maxStakeTokens) external
    onlyOwner {
230     require(pools[_poolId].inited, "Invalid Pool");
231     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
232
233     _updatePoolInfo(_poolId);
234
235     require(pools[_poolId].extension.currentPoolChangeId + 10 >
    poolsChanges[_poolId].length, "Exceed pending changes");
236
```

```
237     PoolChanges memory changes = PoolChanges({
238         applied: false, //
239         updateParamId: UpdateParam.MaxStakeTokens,
240         updateParamValue: _maxStakeTokens,
241         timestamp: block.timestamp,
242         blockNumber: block.number
243     });
244     poolsChanges[_poolId].push(changes);
245 
246     emit PoolUpdated("PoolUpdated", _poolId, pools[_poolId], changes,
    block.number + pools[_poolId].updateDelay);
247 }
```

Listing 12.1 The *updateMaxStakeTokens*, one of the functions that execute the *_updatePoolInfo* function incorrectly

Recommendations

We recommend **updating the functions *updateMaxStakeTokens*, *updateRewardTokensPerBlock*, and *updateEndBlock***.

In code snippet 12.2, for example, the *updateMaxStakeTokens* function was updated **by executing the *_updatePoolInfo* function in L231 before checking the active block in L233**.

VucaStaking.sol

```
229 function updateMaxStakeTokens(uint16 _poolId, uint256 _maxStakeTokens) external
    onlyOwner {
230     require(pools[_poolId].inited, "Invalid Pool");
231     _updatePoolInfo(_poolId);
232 
233     require(block.number + pools[_poolId].updateDelay < pools[_poolId].endBlock,
    "Exceed Blocks");
234 
235     require(pools[_poolId].extension.currentPoolChangeId + 10 >
    poolsChanges[_poolId].length, "Exceed pending changes");
236 
237     PoolChanges memory changes = PoolChanges({
238         applied: false, //
239         updateParamId: UpdateParam.MaxStakeTokens,
240         updateParamValue: _maxStakeTokens,
241         timestamp: block.timestamp,
242         blockNumber: block.number
243     });
244     poolsChanges[_poolId].push(changes);
245 
246     emit PoolUpdated("PoolUpdated", _poolId, pools[_poolId], changes,
    block.number + pools[_poolId].updateDelay);
```

```
247 }
```

Listing 12.2 The improved *updateMaxStakeTokens* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue according to our suggestion.

No. 13	Possibly Stealing All Pools' Staking and Reward Tokens		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 312 - 332 and 335 - 346*		

Detailed Issue

We discovered that both functions ***retrieveReward*** (code snippet 13.1) and ***withdrawERC20*** (code snippet 13.2) **can enable an attacker with a compromised owner account to steal all staking and reward tokens from all staking pools**.

Please find the root cause of each function as follows.

- ***retrieveReward*** function (**Issue #2 - Depending On Incorrect Reward Token Balance #1**)
- ***withdrawERC20*** function (**Issue #1 - Potentially Draining Pools' Reward Tokens**)

Consider the following attack steps.

1. An attacker with a compromised owner account **creates a dummy short-lived pool by setting the pool's reward token as the staking token of the target pool**
2. The attacker **creates another dummy short-lived pool and sets the pool's reward token as the reward token of the target pool**
3. The attacker **waits for both dummy pools to reach their staking period**
4. The attacker **executes either the *retrieveReward* or *withdrawERC20* function on both dummy pools** to drain all staking and reward tokens from the target pool
5. The attacker **performs the attack steps #1 - #4 on other staking pools** to drain all tokens locked in the *VucaStaking* contract

VucaStaking.sol

```
312 function retrieveReward(
313     uint16 _poolId,
314     address _to,
315     uint256 _amount
316 ) external onlyOwner {
317     _updatePoolInfo(_poolId);
318     Pool memory pool = pools[_poolId];
319     require(pool.endBlock <= block.number, "Staking active");
320 
321     _updatePoolRewards(_poolId, block.number);
322     pool = pools[_poolId];
323 
324     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
325     uint256 rewardsWithdrew = pool.rewardsWithdrew /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
326     uint256 contractBalance = IERC20(pool.rewardToken).balanceOf(address(this));
327 
328     // maximum amount withdrawal = balance - max claimable
329     require(_amount + totalUserRewards <= contractBalance + rewardsWithdrew);
330 
331     IERC20(pool.rewardToken).transfer(_to, _amount);
332 }
```

Listing 13.1 The *retrieveReward* function that could drain all reward tokens

VucaStaking.sol

```
335 function withdrawERC20(
336     uint16 _poolId,
337     address _to,
338     uint256 _amount
339 ) external onlyOwner {
340     _updatePoolInfo(_poolId);
341     Pool memory pool = pools[_poolId];
342     require(pool.endBlock <= block.number, "Staking active");
343     require(pool.tokensStaked == 0, "Not allowed");
344 
345     IERC20(pool.rewardToken).transfer(_to, _amount);
346 }
```

Listing 13.2 The *withdrawERC20* function that could drain all reward tokens

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing both *retrieveReward* and *withdrawERC20* functions and their related subsystems **to track each pool's *staking* and *reward tokens* isolatedly**.

Also, please refer to **Issues #1 (Potentially Draining Pools' Reward Tokens)** and **#2 (Depending On Incorrect Reward Token Balance #1)** for more details.

Reassessment

The *Vega Investment Group* team fixed this issue by reworking the *createPool* function (L191 in the code snippet below) to **allow the creation of only one staking pool for each *VucaStaking* contract**.

VucaStaking.sol

```
179 function createPool(
180     address _rewardToken,
181     address _stakeToken,
182     uint256 _maxStakeTokens,
183     uint256 _startBlock,
184     uint256 _endBlock,
185     uint256 _rewardTokensPerBlock,
186     uint32 _updateDelay
187 ) external onlyOwner {
188     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
189     require(_rewardToken != address(0), "Invalid reward token");
190     require(_stakeToken != address(0), "Invalid staking token");
191     require(currentPoolId == 0, "Staking pool was already created");
192
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
200
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204
205     emit PoolCreated(1, currentPoolId, pools[currentPoolId], block.number);
206     currentPoolId += 1;
207 }
```

Listing 13.3 The *createPool* function
that allows the creation of only one staking pool

No. 14	Incorrect Calculation Of Withdrawable Pool Rewards #1		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 81 - 111, 114 - 125, and 211 - 233*		

Detailed Issue

We found an incorrect calculation of the withdrawable pool rewards when invoking the *getRewardsWithdrawable* function (code snippet 14.1).

Specifically, the *getRewardsWithdrawable* function would ask for the *pool* object with up-to-date parameters (L115 in code snippet 14.1) from the *getLatestPoolInfo* function (code snippet 14.2). At this point, **we noticed that the *getLatestPoolInfo* function could return the *pool* object with incorrect parameters.**

Consequently, the incorrect *pool*'s parameters would eventually make the calculation of the pool rewards withdrawable (L122 - 124 in code snippet 14.1) returned by the *getRewardsWithdrawable* function to be incorrect.

VucaStaking.sol

```
114 function getRewardsWithdrawable(uint16 _poolId) public view returns (uint256) {
115     Pool memory pool = getLatestPoolInfo(_poolId);
116
117     uint256 contractBalance = IERC20(pool.rewardToken).balanceOf(address(this));
118     if (pool.endBlock > block.number || contractBalance == 0) {
119         return 0;
120     }
121
122     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
123     uint256 rewardsWithdrew = pool.rewardsWithdrew /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
124     return contractBalance + rewardsWithdrew - totalUserRewards;
125 }
```

Listing 14.1 The *getRewardsWithdrawable* function that incorrectly calculates the withdrawable pool rewards

The root cause of this issue is that the *getLatestPoolInfo* function mistakenly applies *pool changes* in memory (L98 - 103 and L107 - 108 in code snippet 14.2), instead of the contract storage.

As a result, when the *getLatestPoolInfo* function executes the *getPoolRewardsCheckpoint* function (L98 and L107 in code snippet 14.2), the *getPoolRewardsCheckpoint* function would return incorrectly computed parameters *pool.accumulatedRewardsPerShare*, *pool.lastRewardedBlock*, and *rewards*.

That is because the *getPoolRewardsCheckpoint* function would load the *pool* object from the contract storage (L220 in code snippet 14.3), which is a different state variable section to the one updated in the memory of the *getLatestPoolInfo* function.

In other words, the *getPoolRewardsCheckpoint* function would always load the outdated *pool* object from the contract storage, leading to the incorrect calculation of the returned parameters *pool.accumulatedRewardsPerShare*, *pool.lastRewardedBlock*, and *rewards* (L222 - 232 in code snippet 14.3).

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
```

```
109
110     return pool;
111 }
```

Listing 14.2 The *getLatestPoolInfo* function
that mistakenly applies *pool changes* in memory

VucaStaking.sol

```
211 function getPoolRewardsCheckpoint(uint16 _poolId, uint256 _blockNumber)
212     public
213     view
214     returns (
215         uint256 accumulatedRewardsPerShare,
216         uint256 lastRewardedBlock,
217         uint256 rewards
218     )
219 {
220     Pool memory pool = pools[_poolId];
221
222     uint256 floorBlock = _blockNumber <= pool.endBlock ? _blockNumber :
    pool.endBlock;
223
224     uint256 blocksSinceLastReward;
225     if (floorBlock >= pool.lastRewardedBlock) {
226         blocksSinceLastReward = floorBlock - pool.lastRewardedBlock;
227     }
228     rewards = blocksSinceLastReward * pool.rewardTokensPerBlock;
229     if (pool.tokensStaked > 0) {
230         accumulatedRewardsPerShare = pool.accumulatedRewardsPerShare + (rewards
    / pool.tokensStaked);
231     }
232     lastRewardedBlock = floorBlock;
233 }
```

Listing 14.3 The *getPoolRewardsCheckpoint* function
that loads the outdated *pool* object from the contract storage

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the associated functions **by ensuring that the functions must refer to the *same state variable section***.

Reassessment

The *Vega Investment Group* team redesigned and reimplemented a new rewarding subsystem.

As a result, the *getRewardsWithdrawable*, *getLatestPoolInfo*, and *getPoolRewardsCheckpoint* functions were removed from the *VucaStaking* contract. Hence, this issue was fixed.

No. 15	Depending On Incorrect Reward Token Balance #2		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 114 - 125*		

Detailed Issue

We detected that the *getRewardsWithdrawable* function depends on the incorrect reward token balance (L117 in the code snippet below), **leading to the return of an incorrect maximum withdrawable reward for the specified pool.**

Consider the following two scenarios to exploit the issue.

1. **If the staking token (*pool.stakeToken*) is the same as the reward token (*pool.rewardToken*) for a pool, and then there are some stakings from users and the staking period of that pool is ended.**

   Then the *if* statement (L118 - 120) would be bypassed (*without concerning that the balance of funds could also be the staked tokens*).

2. **If the reward token (*pool.rewardToken*) of one pool is the same token utilized by another pool that is using the token as a staking or reward token.**

   Then the *if* statement (L118 - 120) would be bypassed.

From the exploit scenarios above, if one condition is met, the *if* statement (L118 - 120) would be bypassed. Later, **the *getRewardsWithdrawable* function would return an incorrect maximum withdrawable reward for the specified pool as the returned reward amount could be the shared funds from multiple pools (or even the same pool with the same staking and reward tokens)**.

VucaStaking.sol

```
114 function getRewardsWithdrawable(uint16 _poolId) public view returns (uint256) {
115     Pool memory pool = getLatestPoolInfo(_poolId);
116
117     uint256 contractBalance = IERC20(pool.rewardToken).balanceOf(address(this));
118     if (pool.endBlock > block.number || contractBalance == 0) {
119         return 0;
120     }
121
122     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
123     uint256 rewardsWithdrew = pool.rewardsWithdrew /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
124     return contractBalance + rewardsWithdrew - totalUserRewards;
125 }
```

Listing 15.1 The *getRewardsWithdrawable* function
that depends on the incorrect reward token balance

Moreover, **we also discovered that the formula for calculating the returned reward is incorrect due to improperly relying on the reward token balance** (L124).

Consider the formula being used by the *getRewardsWithdrawable* function (L124).

rewardsWithdrawable = contractBalance + rewardsWithdrew - totalUserRewards

As the ***contractBalance*** (the reward token balance) could indicate the total balance aggregated from multiple pools, the use of this incorrect balance could result in an incorrectly returned reward.

For example, if *contractBalance* = 100 (aggregated from multiple pools), *rewardsWithdrew* = 50 (for the specified pool), *totalUserRewards* = 80 (for the specified pool). Then the *rewardsWithdrawable* could be computed as **100 + 50 - 80 = 70 (not 30)**.

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the new rewarding subsystem **to track each pool's *staking* and *reward tokens* separately**.

Reassessment

The *getRewardsWithdrawable* function was removed from the *VucaStaking* contract. Hence, this issue was fixed.

No. 16	Lack Of Guaranteeing Pool State Consistency		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 162 - 164 and 195 - 197*		

Detailed Issue

We noticed that the functions ***emergencyWithdraw*** (L162 - 164) and ***unStake*** (L195 - 197) in the code snippet below might not maintain or guarantee the pool's state consistency.

More specifically, in case the ***pool.tokensStaked < amount***, the pool's ***tokensStaked*** parameter would not be updated, leading to a state inconsistency issue to the pool.

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
154     _updatePoolInfo(_poolId);
155     Pool storage pool = pools[_poolId];
156     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
157     uint256 amount = staking.amount;
158     require(staking.amount > 0, "Insufficient funds");
159 
160     _updatePoolRewards(_poolId, block.number);
161     // Update pool
162     if (pool.tokensStaked >= amount) {
163         pool.tokensStaked -= amount;
164     }
165 
166     staking.amount = 0;
167 
168     // Withdraw tokens
169     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
170 
171     emit StakingChanged(msg.sender, _poolId, pool, staking);
172 
173     // Update staker
174     staking.accumulatedRewards = 0;
175     staking.minusRewards = 0;
176 }
```

```
    // (...SNIPPED...)

179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock <= block.number, "Staking active");
183 
184     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
185     uint256 amount = staking.amount;
186     require(staking.amount > 0, "Insufficient funds");
187 
188     _updatePoolRewards(_poolId, block.number);
189     // Pay rewards
190     uint256 rewards = getRewards(_poolId, msg.sender);
191     IERC20(pool.rewardToken).transfer(msg.sender, rewards);
192 
193     // Update pool
194     pool.rewardsWithdrew += getRawRewards(_poolId, msg.sender);
195     if (pool.tokensStaked >= amount) {
196         pool.tokensStaked -= amount;
197     }
198 
199     // Withdraw tokens
200     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
201 
202     emit StakingChanged(msg.sender, _poolId, pool, staking);
203 
204     // Update staker
205     staking.accumulatedRewards = 0;
206     staking.minusRewards = 0;
207     staking.amount = 0;
208 }
```

Listing 16.1 The *emergencyWithdraw* and *unStake* functions

Recommendations

We recommend updating both the functions ***emergencyWithdraw*** (L162) and ***unStake*** (L193) as presented in the code snippet below. In other words, both functions would revert transactions if the ***pool.tokensStaked < amount*** (incurring state inconsistency).

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
        // (...SNIPPED...)
162     pool.tokensStaked -= amount;
        // (...SNIPPED...)
174 }


    // (...SNIPPED...)
177 function unStake(uint16 _poolId) external {
        // (...SNIPPED...)
193     pool.tokensStaked -= amount;
        // (...SNIPPED...)
204 }
```

Listing 16.2 The improved *emergencyWithdraw* and *unStake* functions

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue as per our recommendation.

No. 17	Usage Of Unsafe Token Transfer Functions		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 149, 169, 191, 200, 331, and 345*		

Detailed Issue

We found some usage of the *ERC20*'s *transfer* and *transferFrom* functions that are providing unsafe token transfers (e.g., the *transfer* function in L169 in the code snippet below) as follows.

1. In the ***stake* function** (***L149** in VucaStaking.sol*)
2. In the ***emergencyWithdraw* function** (***L169** in VucaStaking.sol*)
3. In the ***unStake* function** (***L191** and **L200** in VucaStaking.sol*)
4. In the ***retrieveReward* function** (***L331** in VucaStaking.sol*)
5. In the ***withdrawERC20* function** (***L345** in VucaStaking.sol*)

The use of unsafe functions could lead to unexpected token transfer errors.

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
154     _updatePoolInfo(_poolId);
155     Pool storage pool = pools[_poolId];
156     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
157     uint256 amount = staking.amount;
158     require(staking.amount > 0, "Insufficient funds");
159 
160     _updatePoolRewards(_poolId, block.number);
161     // Update pool
162     if (pool.tokensStaked >= amount) {
163       pool.tokensStaked -= amount;
164     }
165 
166     staking.amount = 0;
```

```
167
168     // Withdraw tokens
169     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
170
171     emit StakingChanged(msg.sender, _poolId, pool, staking);
172
173     // Update staker
174     staking.accumulatedRewards = 0;
175     staking.minusRewards = 0;
176 }
```

Listing 17.1 The *emergencyWithdraw*,
one of the functions that use an unsafe *transfer* function

Recommendations

We recommend applying the safer functions as follows.

- *ERC20*'s *transfer* function -> ***SafeERC20*'s *safeTransfer* function**
- *ERC20*'s *transferFrom* function -> ***SafeERC20*'s *safeTransferFrom* function**

Reassessment

This issue was fixed by employing the *safeTransfer* and *safeTransferFrom* functions according to our recommendation.

No. 18	Removal Recommendation For Mock Function		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 302 - 309*		

Detailed Issue

We found the mock function named *updateChangesDelayBlocks* (the code snippet below) that should not be put in production. **This mock function allows an owner to update the *updateDelay* parameter (L305) of any staking pools which could conflict with the protocol design.**

VucaStaking.sol

```
301 /* @Dev only, remove in prod */
302 function updateChangesDelayBlocks(uint16 _poolId, uint32 _blocks) external
    onlyOwner {
303     require(pools[_poolId].inited, "Invalid Pool");
304
305     pools[_poolId].updateDelay = _blocks;
306     PoolChanges memory changes;
307
308     emit PoolUpdated(_poolId, pools[_poolId], changes, block.number);
309 }
```

Listing 18.1 The mock function *updateChangesDelayBlocks*

Recommendations

We recommend removing the mock function *updateChangesDelayBlocks* from the *VucaStaking* contract.

Reassessment

The mock function *updateChangesDelayBlocks* was removed from the *VucaStaking* contract to fix this issue.

No. 19	Possibly Permanent Ownership Removal		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*@openzeppelin/contracts/access/Ownable.sol*		
Locations	*Ownable.sol L: 61 - 63*		

Detailed Issue

The *CrownToken* and *VucaStaking* contracts inherit from the *Ownable* abstract contract. The *Ownable* contract implements the *renounceOwnership* function (L61 - 63 in the code snippet below), which can remove the contract's ownership permanently.

If the contract owner mistakenly invokes the *renounceOwnership* function, they will immediately lose ownership of the contract, and this action cannot be undone.

Ownable.sol

```
61  function renounceOwnership() public virtual onlyOwner {
62      _transferOwnership(address(0));
63  }

    // (...SNIPPED...)

78  function _transferOwnership(address newOwner) internal virtual {
79      address oldOwner = _owner;
80      _owner = newOwner;
81      emit OwnershipTransferred(oldOwner, newOwner);
82  }
```

Listing 19.1 The *renounceOwnership* function
that can remove the ownership of the contract permanently

Recommendations

We consider the *renounceOwnership* function risky, and the contract owner should use this function with extra care.

If possible, we recommend removing or disabling this function from the contract. The code snippet below shows an example solution to disabling the associated *renounceOwnership* function.

To remediate this issue, please apply the following code to both the *CrownToken* and *VucaStaking* contracts.

CrownToken.sol

```
16 function renounceOwnership() external override onlyOwner {
17     revert("Ownable: renounceOwnership function is disabled");
18 }
```

Listing 19.2 The disabled *renounceOwnership* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue by disabling the *renounceOwnership* function on the *VucaStaking* contract. For the *CrownToken* contract, the team decided to remove the inheritance from the associated *Ownable* contract.

No. 20	Unsafe Ownership Transfer		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	@openzeppelin/contracts/access/Ownable.sol		
Locations	Ownable.sol L: 69 - 72		

Detailed Issue

The *CrownToken* and *VucaStaking* contracts inherit from the *Ownable* abstract contract. The *Ownable* contract implements the *transferOwnership* function (L69 - 72 in the code snippet below), which can transfer the ownership of the contract from the current owner to another owner.

Ownable.sol

```
69  function transferOwnership(address newOwner) public virtual onlyOwner {
70      require(newOwner != address(0), "Ownable: new owner is the zero address");
71      _transferOwnership(newOwner);
72  }

    // (...SNIPPED...)

78  function _transferOwnership(address newOwner) internal virtual {
79      address oldOwner = _owner;
80      _owner = newOwner;
81      emit OwnershipTransferred(oldOwner, newOwner);
82  }
```

Listing 20.1 The *transferOwnership* function that has the unsafe ownership transfer

From the code snippet above, the address variable *newOwner* (L69) may be incorrectly specified by the current owner by mistake; for example, an address that a new owner does not own was inputted. Consequently, the new owner loses ownership of the contract immediately, and this action is unrecoverable.

Recommendations

We recommend applying the two-step ownership transfer mechanism as shown in the code snippet below.

CrownToken.sol

```
16  function transferOwnership(address _candidateOwner) public override onlyOwner {
17      require(_candidateOwner != address(0), "Ownable: candidate owner is the zero
    address");
18      candidateOwner = _candidateOwner;
19      emit NewCandidateOwner(_candidateOwner);
20  }
21
22  function claimOwnership() external {
23      require(candidateOwner == _msgSender(), "Ownable: caller is not the
    candidate owner");
24      _transferOwnership(candidateOwner);
25      candidateOwner = address(0);
26  }
```

Listing 20.2 The recommended two-step ownership transfer mechanism

This mechanism works as follows.

1. The current owner invokes the *transferOwnership* function by specifying the candidate owner address *_candidateOwner* (L16).
2. The candidate owner proves access to his account and claims the ownership transfer by invoking the *claimOwnership* function (L22)

The recommended mechanism ensures that the ownership of the contract would be transferred to another owner who can access his account only.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

To remediate this issue, please apply the above code to both the *CrownToken* and *VucaStaking* contracts.

Reassessment

The *Vega Investment Group* team fixed this issue by applying the two-step ownership transfer mechanism to the *VucaStaking* contract as per our suggestion. For the *CrownToken* contract, the team decided to remove the inheritance from the associated *Ownable* contract.

No. 21	Recommended Improving Transparency And Trustworthiness Of Privileged Operations		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/VucaStaking.sol* *@openzeppelin/contracts/access/Ownable.sol*		
Locations	*VucaStaking.sol L: 236 - 265, 267 - 275, 277 - 287, 290 - 299, 302 - 309, 312 - 332, and 335 - 346* *Ownable.sol L: 61 - 63 and 69 - 72*		

Detailed Issue

The following lists all owner-privileged setter functions.

1. ***createPool* function** (**L236 - 265** in *VucaStaking.sol*)
2. ***updateMaxStakeTokens* function** (**L267 - 275** in *VucaStaking.sol*)
3. ***updateRewardTokensPerBlock* function** (**L277 - 287** in *VucaStaking.sol*)
4. ***updateEndBlock* function** (**L290 - 299** in *VucaStaking.sol*)
5. ***updateChangesDelayBlocks* function** (**L302 - 309** in *VucaStaking.sol*)
6. ***retrieveReward* function** (**L312 - 332** in *VucaStaking.sol*)
7. ***withdrawERC20* function** (**L335 - 346** in *VucaStaking.sol*)
8. ***renounceOwnership* function** (**L61 - 63** in *Ownable.sol* **for both *CrownToken* and *VucaStaking* contracts**)
9. ***transferOwnership* function** (**L69 - 72** in *Ownable.sol* **for both *CrownToken* and *VucaStaking* contracts**)

Our analysis found that **the *setter functions* listed above can change important states of the *CrownToken* and/or *VucaStaking* contracts which could affect the users' assets**.

For this reason, we consider that those *setter functions* should be improved for transparency and trustworthiness.

Recommendations

We recommend governing the associated setter functions with the ***Multisig***, ***Timelock***, and/or ***DAO (Decentralized Autonomous Organization)*** mechanisms to improve the transparency and trustworthiness of the privileged operations.

Reassessment

This issue was acknowledged by the *Vega Investment Group* team.

No. 22	Users Can Mistakenly Transfer Reward Tokens To Staking Pools		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 209 - 227 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We noticed that **the *depositPoolReward* function (code snippet 22.1) allows anyone to execute it** to transfer reward tokens to a specific pool.

If a user calls this function by mistake, a user would lose his/her funds immediately.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
(10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
amount);
221 
222     pool.extension.totalPoolRewards = totalPoolRewards;
223 
224     PoolChanges memory changes;
225 
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
```

```
    block.number);
227 }
```

Listing 22.1 The *depositPoolReward* function that allows anyone to call it

Recommendations

We recommend **applying the onlyOwner modifier** to the *depositPoolReward* function (L209 in code snippet 22.2).

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public onlyOwner {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
221 
222     pool.extension.totalPoolRewards = totalPoolRewards;
223 
224     PoolChanges memory changes;
225 
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
    block.number);
227 }
```

Listing 22.2 The improved *depositPoolReward* function
that allows only a contract owner to call it

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team acknowledged this issue. However, **the team decided not to fix this issue as applying the *onlyOwner* modifier would remove their flexibility**.

No. 23	Incorrect Calculation Of Withdrawable Pool Rewards #2		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 81 - 111, 114 - 125, and 211 - 233*		

Detailed Issue

We discovered an incorrect calculation of the withdrawable pool rewards when invoking the *getRewardsWithdrawable* function (code snippet 23.1).

Specifically, the *getRewardsWithdrawable* function would ask for the *pool* object with up-to-date parameters (L115 in code snippet 23.1) from the *getLatestPoolInfo* function (code snippet 23.2). At this point, **we noticed that the *getLatestPoolInfo* function could return the *pool* object with the inaccurate parameter *totalUserRewards*.**

Subsequently, the inaccurate parameter *totalUserRewards* would eventually make the calculation of the pool rewards withdrawable (L122 and L124 in code snippet 23.1) returned by the *getRewardsWithdrawable* function to be incorrect.

VucaStaking.sol

```
114 function getRewardsWithdrawable(uint16 _poolId) public view returns (uint256) {
115     Pool memory pool = getLatestPoolInfo(_poolId);
116
117     uint256 contractBalance = IERC20(pool.rewardToken).balanceOf(address(this));
118     if (pool.endBlock > block.number || contractBalance == 0) {
119         return 0;
120     }
121
122     uint256 totalUserRewards = pool.totalUserRewards /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
123     uint256 rewardsWithdrew = pool.rewardsWithdrew /
    (10**IERC20(pool.stakeToken).decimals()) / REWARDS_PRECISION;
124     return contractBalance + rewardsWithdrew - totalUserRewards;
125 }
```

Listing 23.1 The *getRewardsWithdrawable* function that incorrectly calculates the withdrawable pool rewards

In the *getLatestPoolInfo* function, we noticed that if the parameter *rewards* returned by the *getPoolRewardsCheckpoint* function (L98 and L107 in code snippet 23.2) is inaccurate, the inaccurate *rewards* would make the *pool*'s *totalUserRewards* (L99 and L108) inaccurate as well.

Next, we found that **the *getPoolRewardsCheckpoint* function would inaccurately compute the *rewards* parameter (L228 in code snippet 23.3) if there is no staking at the moment of computation (*pool.tokensStaked == 0*).**

One example situation that could trigger this issue is when a staking pool is active but there is no staking yet. The *getPoolRewardsCheckpoint* function would return the parameter *rewards* with a positive value, which is incorrect. That is, the parameter *rewards* should ideally be 0 in that case.

VucaStaking.sol

```
81  function getLatestPoolInfo(uint16 _poolId) public view returns (Pool memory) {
82      Pool memory pool = pools[_poolId];
83
84      uint256 size = poolsChanges[_poolId].length;
85      for (uint256 i; i < size; i++) {
86          PoolChanges memory changes = poolsChanges[_poolId][i];
87
88          if (changes.applied) {
89              continue;
90          }
91
92          uint256 updateAtBlock = changes.blockNumber + pool.updateDelay;
93          if (!(pool.endBlock > updateAtBlock && block.number >= updateAtBlock)) {
94              continue;
95          }
96
97          uint256 rewards;
98          (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, rewards) =
    getPoolRewardsCheckpoint(_poolId, updateAtBlock);
99          pool.totalUserRewards += rewards;
100
101         pool.maxStakeTokens = changes.maxStakeTokens;
102         pool.endBlock = changes.endBlock;
103         pool.rewardTokensPerBlock = changes.rewardTokensPerBlock;
104     }
105
106     uint256 _rewards;
107     (pool.accumulatedRewardsPerShare, pool.lastRewardedBlock, _rewards) =
    getPoolRewardsCheckpoint(_poolId, block.number);
108     pool.totalUserRewards += _rewards;
109
110     return pool;
111 }
```

Listing 23.2 The *getLatestPoolInfo* function
that incorrectly calculates the staking pool's *totalUserRewards* parameter

VucaStaking.sol

```
211 function getPoolRewardsCheckpoint(uint16 _poolId, uint256 _blockNumber)
212     public
213     view
214     returns (
215         uint256 accumulatedRewardsPerShare,
216         uint256 lastRewardedBlock,
217         uint256 rewards
218     )
219 {
220     Pool memory pool = pools[_poolId];
221 
222     uint256 floorBlock = _blockNumber <= pool.endBlock ? _blockNumber :
    pool.endBlock;
223 
224     uint256 blocksSinceLastReward;
225     if (floorBlock >= pool.lastRewardedBlock) {
226         blocksSinceLastReward = floorBlock - pool.lastRewardedBlock;
227     }
228     rewards = blocksSinceLastReward * pool.rewardTokensPerBlock;
229     if (pool.tokensStaked > 0) {
230         accumulatedRewardsPerShare = pool.accumulatedRewardsPerShare + (rewards
    / pool.tokensStaked);
231     }
232     lastRewardedBlock = floorBlock;
233 }
```

Listing 23.3 The *getPoolRewardsCheckpoint* function that would inaccurately compute
the parameter *rewards* if there is no staking at the computation moment

Recommendations

We recommend updating the *getPoolRewardsCheckpoint* function like the code snippet below.

The *getPoolRewardsCheckpoint* function would compute the parameter *rewards* if and only if there must be any staking at the computation moment (L229). If there is no staking, the parameter *rewards* would be 0.

VucaStaking.sol

```
211 function getPoolRewardsCheckpoint(uint16 _poolId, uint256 _blockNumber)
212     public
213     view
214     returns (
215         uint256 accumulatedRewardsPerShare,
216         uint256 lastRewardedBlock,
217         uint256 rewards
218     )
219 {
220     Pool memory pool = pools[_poolId];
221 
222     uint256 floorBlock = _blockNumber <= pool.endBlock ? _blockNumber :
    pool.endBlock;
223 
224     if (pool.tokensStaked > 0) {
225         uint256 blocksSinceLastReward;
226         if (floorBlock >= pool.lastRewardedBlock) {
227             blocksSinceLastReward = floorBlock - pool.lastRewardedBlock;
228         }
229         rewards = blocksSinceLastReward * pool.rewardTokensPerBlock;
230         accumulatedRewardsPerShare = pool.accumulatedRewardsPerShare + (rewards
    / pool.tokensStaked);
231     }
232     lastRewardedBlock = floorBlock;
233 }
```

Listing 23.4 The improved *getPoolRewardsCheckpoint* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The associated functions *getRewardsWithdrawable*, *getLatestPoolInfo*, and *getPoolRewardsCheckpoint* were removed from the *VucaStaking* contract. Hence, this issue was closed.

No. 24	Possibly Unstaking Or Retrieving Reward Tokens Before Staking Period Ends		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 182, 319, and 342*		

Detailed Issue

We found **some nuance validation mistakes on the staking pool's *endBlock* parameter** on the following functions.

1. In the ***unStake* function** (***L182** in VucaStaking.sol*)
2. In the ***retrieveReward* function** (***L319** in VucaStaking.sol*)
3. In the ***withdrawERC20* function** (***L342** in VucaStaking.sol*)

For instance, the ***require(pool.endBlock <= block.number, "Staking active");*** statement in L182 in the code snippet below. The root cause is that the ***require*** statement includes the case that the ***block.number == pool.endBlock***.

Consequently, each staking pool can be unstaked, or retrieved its *reward tokens* via the *unStake*, *retrieveReward*, and *withdrawERC20* functions before the staking period ends.

VucaStaking.sol

```
179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock <= block.number, "Staking active");

        // (...SNIPPED...)

208 }
```

Listing 24.1 The *unStake*, one of the functions that are affected to the issue

Recommendations

We recommend revising the associated ***require*** statements (***L182***, ***L319***, and ***L342*** in *VucaStaking.sol*) by excluding the case of the ***block.number == pool.endBlock*** similar to L182 in the code snippet below.

VucaStaking.sol

```
179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock < block.number, "Staking active");

        // (...SNIPPED...)

208 }
```

Listing 24.2 The improved *unStake* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed according to our recommendation.

No. 25	Recommended Event Emissions For Transparency And Traceability		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 312 - 332, 335 - 346, 349 - 371, and 373 - 385*		

Detailed Issue

We consider operations of the following state-changing functions important and require proper event emissions for improving transparency and traceability.

- ***retrieveReward*** **function** (**L312 - 332** in *VucaStaking.sol*)
- ***withdrawERC20*** **function** (**L335 - 346** in *VucaStaking.sol*)
- ***_updatePoolInfo*** **function** (**L349 - 371** in *VucaStaking.sol*)
- ***_updatePoolRewards*** **function** (**L373 - 385** in *VucaStaking.sol*)

Recommendations

We recommend **emitting relevant events** on the associated functions to improve transparency and traceability.

Reassessment

The *retrieveReward* function was improved to emit a proper event, whereas the *Vega Investment Group* team removed the *withdrawERC20* function. For the *_updatePoolInfo* and *_updatePoolRewards* functions, the team decided not to emit an event, nevertheless.

For this reason, this issue was considered *partially fixed*.

No. 26	Compiler Is Not Locked To Specific Version		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CrownToken.sol* *contracts/VucaStaking.sol*		
Locations	*CrownToken.sol L: 2* *VucaStaking.sol L: 2*		

Detailed Issue

The *CrownToken* and *VucaStaking* smart contracts should be deployed with the compiler version used in the development and testing process.

The compiler version that is not strictly locked via the *pragma* statement may make the contract incompatible against unforeseen circumstances.

An example code that is not locked to a specific version (e.g., using `>=` or `^` directive) is shown below.

CrownToken.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.15;
```

Listing 26.1 An example code that is not locked to a specific version

Recommendations

We recommend locking the *pragma* version like the example code snippet below.

```
pragma solidity 0.8.0;
// or
pragma solidity =0.8.0;

contract SemVerFloatingPragmaFixed {
}
```

Reference: *https://swcregistry.io/docs/SWC-103*

Reassessment

The *Vega Investment Group* team fixed this issue by locking the *pragma* version to v0.8.17.

No. 27	Compiler May Be Susceptible To Publicly Disclosed Bugs		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/CrownToken.sol* *contracts/VucaStaking.sol*		
Locations	*CrownToken.sol L: 2* *VucaStaking.sol L: 2*		

Detailed Issue

The *CrownToken* and *VucaStaking* smart contracts use an outdated Solidity compiler version (v0.8.15) which may be susceptible to publicly disclosed vulnerabilities. The latest compiler patch version is 0.8.17, which contains the list of known bugs as the following link:

https://docs.soliditylang.org/en/v0.8.17/bugs.html

The known bugs may not directly lead to the vulnerability, but it may increase an opportunity to trigger some attacks further.

An example smart contract that does not use the latest patch version is shown below.

CrownToken.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.15;
```

Listing 27.1 An example smart contract that does not use the latest patch version (v0.8.17)

Recommendations

We recommend using the latest patch version, v0.8.17, that fixes all known bugs.

Reassessment

The *Vega Investment Group* team fixed this issue by applying the patch version v0.8.17.

No. 28	Lack Of Applying Pool Changes		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 209 - 227 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We discovered that the *depositPoolReward* function does not apply active pool changes before calculating the parameters *rewardTokenPerBlock* (L213 in code snippet 28.1) and *totalPoolRewards* (L214).

If there are active pool changes for updating the parameters ***pool.endBlock*** and ***pool.rewardTokensPerBlock*** pending in the queue, the resulting computed parameters *rewardTokenPerBlock* (L213) and *totalPoolRewards* (L214) would be incorrect, leading to depositing an incorrect amount of reward tokens to a staking pool.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
221 
222     pool.extension.totalPoolRewards = totalPoolRewards;
```

```
223
224     PoolChanges memory changes;
225
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
    block.number);
227 }
```

Listing 28.1 The *depositPoolReward* function
that does not apply active pool changes

Recommendations

We recommend updating the *depositPoolReward* function by **invoking the *_updatePoolInfo* function (like L212 in code snippet 28.2) to apply all active pool changes** before calculating the parameters *rewardTokenPerBlock* (L214) and *totalPoolRewards* (L215).

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212     _updatePoolInfo(_poolId);
213
214     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
215     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
216
217     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
218
219     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
220
221     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
222
223     pool.extension.totalPoolRewards = totalPoolRewards;
224
225     PoolChanges memory changes;
226
227     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
    block.number);
228 }
```

Listing 28.2 The improved *depositPoolReward* function that applies all active pool changes
before calculating the parameters *rewardTokenPerBlock* and *totalPoolRewards*

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team reworked the *depositPoolReward* function, as per the code snippet below, which also fixed this issue.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId, uint256 _amount) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212     require(_amount > 0, "Invalid amount");
213     _updatePoolInfo(_poolId);
214 
215     pool.extension.totalPoolRewards += _amount;
216 
217     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    _amount);
218 
219     PoolChanges memory changes;
220     emit PoolUpdated(2, _poolId, pools[_poolId], changes, block.number);
221 }
```

Listing 28.3 The reworked *depositPoolReward* function

No. 29	Incorrectly Calculating Total Pool Rewards		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 209 - 227 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We found that the formula used for calculating the parameter *totalPoolRewards* (L214 in code snippet 29.1) is incorrect.

In a word, the formula includes one more block (***pool.endBlock - pool.startBlock + 1***) than the actual value (***pool.endBlock - pool.startBlock***), leading to an incorrect pool staking period.

As a result, the incorrect staking period would require an owner to deposit more reward tokens than the actual amount.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
221 
```

```
222     pool.extension.totalPoolRewards = totalPoolRewards;
223
224     PoolChanges memory changes;
225
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
    block.number);
227 }
```

Listing 29.1 The *depositPoolReward* function that incorrectly calculates the parameter *totalPoolRewards*

Recommendations

We recommend updating the associated formula (like L214 in code snippet 29.2) to correct the staking period.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock);
215
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
217
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
221
222     pool.extension.totalPoolRewards = totalPoolRewards;
223
224     PoolChanges memory changes;
225
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
    block.number);
227 }
```

Listing 29.2 The improved *depositPoolReward* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team reworked the *depositPoolReward* function, as per the code snippet below, which also fixed this issue.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId, uint256 _amount) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212     require(_amount > 0, "Invalid amount");
213     _updatePoolInfo(_poolId);
214 
215     pool.extension.totalPoolRewards += _amount;
216 
217     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    _amount);
218 
219     PoolChanges memory changes;
220     emit PoolUpdated(2, _poolId, pools[_poolId], changes, block.number);
221 }
```

Listing 29.3 The reworked *depositPoolReward* function

No. 30	Incorrectly Calculating User's Pool Rewards		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 82 - 89 and 92 - 96 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

The ***getRawRewards*** **(L82 - 89 in code snippet 30.1) and** ***getRewards*** **(L92 - 96) are** ***public*** **functions** that an external caller can execute to get the user's pool rewards.

However, we found that if the functions are externally called, they could return incorrect pool rewards because they could operate on an outdated *pool* object (L84 in code snippet 30.1). In other words, **if there are active pending pool changes in the queue, the *pool* object loaded by the *getRawRewards* function in L84 would not be updated**.

As a result, the execution of the *_getPoolRewards* function (code snippet 30.2) in L86 in code snippet 30.1 would return an incorrectly computed *pool* object, **making the calculation in L88 to be incorrect**.

VucaStaking.sol

```
81  // rewards w/o adjustment
82  function getRawRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
83      Staking memory staking = stakingUsersInfo[_poolId][_account];
84      Pool memory pool = pools[_poolId];
85
86      pool = _getPoolRewards(pool, block.number);
87
88      return staking.accumulatedRewards + (staking.amount *
    pool.accumulatedRewardsPerShare) - staking.minusRewards;
89  }
90
91  // rewards with adjustment
92  function getRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
```

```
93      uint256 rawRewards = getRawRewards(_poolId, _account);
94
95      return rawRewards / (10**IERC20Helper(pools[_poolId].stakeToken).decimals())
    / REWARDS_PRECISION;
96  }
```

Listing 30.1 The *getRawRewards* and *getRewards* functions
that could return the incorrect user's pool rewards

VucaStaking.sol

```
361 function _getPoolRewards(Pool memory _pool, uint256 _blockNumber) internal pure
    returns (Pool memory) {
362     uint256 floorBlock = _blockNumber <= _pool.endBlock ? _blockNumber :
    _pool.endBlock;
363
364     if (_pool.tokensStaked == 0) {
365         _pool.lastRewardedBlock = floorBlock;
366         return _pool;
367     }
368
369     uint256 blocksSinceLastReward;
370     if (floorBlock >= _pool.lastRewardedBlock) {
371         blocksSinceLastReward = floorBlock - _pool.lastRewardedBlock;
372     }
373     uint256 rewards = blocksSinceLastReward * _pool.rewardTokensPerBlock;
374     _pool.accumulatedRewardsPerShare = _pool.accumulatedRewardsPerShare +
    (rewards / _pool.tokensStaked);
375     _pool.lastRewardedBlock = floorBlock;
376     _pool.extension.totalUserRewards += rewards;
377
378     return _pool;
379 }
```

Listing 30.2 The *_getPoolRewards* function that could incorrectly calculate
the *pool* object using the outdated pool's info

Recommendations

We recommend two possible solutions.

1. **Changing the access visibility of the *getRawRewards* and *getRewards* functions to *private* or *internal*** to allow only the *VucaStaking* contract's functions to execute them internally.

2. **Updating the *getRawRewards* function by simulating the up-to-date *pool* object in memory** like the code snippets 30.3 and 30.4 below.

   The *getRawRewards* function would execute the *_updatePoolInfoInMemory* function (L85 in code snippet 30.3) to get the up-to-date *pool* object, and then pass the *pool* object to the *_getPoolRewards* function (L86).

   Code snippet 30.4 presents **the functions *_updatePoolInfoInMemory* (L98 - 122) and *_updatePoolRewardsInMemory* (L124 - 131) that simulate the up-to-date *pool* object in memory**.

VucaStaking.sol

```
81  // rewards w/o adjustment
82  function getRawRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
83      Staking memory staking = stakingUsersInfo[_poolId][_account];
84
85      Pool memory pool = _updatePoolInfoInMemory(_poolId);
86      pool = _getPoolRewards(pool, block.number);
87
88      return staking.accumulatedRewards + (staking.amount *
    pool.accumulatedRewardsPerShare) - staking.minusRewards;
89  }
90
91  // rewards with adjustment
92  function getRewards(uint16 _poolId, address _account) public view returns
    (uint256) {
93      uint256 rawRewards = getRawRewards(_poolId, _account);
94
95      return rawRewards / (10**IERC20Helper(pools[_poolId].stakeToken).decimals())
    / REWARDS_PRECISION;
96  }
```

Listing 30.3 The improved *getRawRewards* and *getRewards* functions that calculate the user's pool rewards using the in-memory up-to-date *pool* object

VucaStaking.sol

```
98  function _updatePoolInfoInMemory(uint16 _poolId) internal view returns (Pool
    memory _pool) {
99      _pool = pools[_poolId];
100
101     uint256 size = poolsChanges[_poolId].length;
102     uint256 i = _pool.extension.currentPoolChangeId;
103     for (; i < size; i++) {
104         PoolChanges memory changes = poolsChanges[_poolId][i];
105
106         uint256 updateAtBlock = changes.blockNumber + _pool.updateDelay;
107         if (!(_pool.endBlock > updateAtBlock && block.number >= updateAtBlock))
    {
108             break;
109         }
110
111         _pool = _updatePoolRewardsInMemory(_pool, updateAtBlock);
112         if (changes.updateParamId == UpdateParam.MaxStakeTokens) {
113             _pool.maxStakeTokens = changes.updateParamValue;
114         } else if (changes.updateParamId == UpdateParam.EndBlock) {
115             _pool.endBlock = changes.updateParamValue;
116         } else if (changes.updateParamId == UpdateParam.RewardTokensPerBlock) {
117             _pool.rewardTokensPerBlock = changes.updateParamValue;
118         }
119         changes.applied = true;
120     }
121     _pool.extension.currentPoolChangeId = i;
122 }
123
124 function _updatePoolRewardsInMemory(Pool memory _pool, uint256 _blockNumber)
    internal view returns (Pool memory) {
125     Pool memory newPool = _getPoolRewards(_pool, _blockNumber);
126
127     _pool.accumulatedRewardsPerShare = newPool.accumulatedRewardsPerShare;
128     _pool.lastRewardedBlock = newPool.lastRewardedBlock;
129     _pool.extension.totalUserRewards = newPool.extension.totalUserRewards;
130     return _pool;
131 }
```

Listing 30.4 The *_updatePoolInfoInMemory* and *_updatePoolRewardsInMemory* functions in which simulate the up-to-date *pool* object in memory

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue by **changing the access visibility of the *getRawRewards* and *getRewards* functions to *internal***.

No. 31	Lack Of Proper Input Sanitization Check		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 180 - 207 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We noticed that the *createPool* function (code snippet 31.1) **lacks a proper sanitization check on the *_updateDelay* parameter**.

If a staking pool is created with an invalid value of the *_updateDelay* parameter, there is no solution for an owner to update that pool parameter in production.

VucaStaking.sol

```
180 function createPool(
181     address _rewardToken,
182     address _stakeToken,
183     uint256 _maxStakeTokens,
184     uint256 _startBlock,
185     uint256 _endBlock,
186     uint256 _rewardTokensPerBlock,
187     uint32 _updateDelay
188 ) external onlyOwner {
189     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
190     require(_rewardToken != address(0), "Invalid reward token");
191     require(_stakeToken != address(0), "Invalid staking token");
192 
193     pools[currentPoolId].inited = true;
194     pools[currentPoolId].rewardToken = _rewardToken;
195     pools[currentPoolId].stakeToken = _stakeToken;
196 
197     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
198     pools[currentPoolId].startBlock = _startBlock;
199     pools[currentPoolId].endBlock = _endBlock;
```

```
200
201     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
202     pools[currentPoolId].lastRewardedBlock = _startBlock;
203     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
204
205     emit PoolCreated("PoolCreated", currentPoolId, pools[currentPoolId],
    block.number);
206     currentPoolId += 1;
207 }
```

Listing 31.1 The *createPool* function that
lacks a proper sanitization check on the *_updateDelay* parameter

Recommendations

We recommend updating the *createPool* function by **adding the proper sanitization check** similar to L192 in the code snippet below.

VucaStaking.sol

```
180 function createPool(
181     address _rewardToken,
182     address _stakeToken,
183     uint256 _maxStakeTokens,
184     uint256 _startBlock,
185     uint256 _endBlock,
186     uint256 _rewardTokensPerBlock,
187     uint32 _updateDelay
188 ) external onlyOwner {
189     require(_startBlock > block.number && _startBlock < _endBlock, "Invalid
    start/end block");
190     require(_rewardToken != address(0), "Invalid reward token");
191     require(_stakeToken != address(0), "Invalid staking token");
192     require(_updateDelay >= MIN_UPDATE_DELAY, "Invalid update delay");
193
194     pools[currentPoolId].inited = true;
195     pools[currentPoolId].rewardToken = _rewardToken;
196     pools[currentPoolId].stakeToken = _stakeToken;
197
198     pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
199     pools[currentPoolId].startBlock = _startBlock;
200     pools[currentPoolId].endBlock = _endBlock;
201
202     pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20Helper(_stakeToken).decimals()) * REWARDS_PRECISION;
203     pools[currentPoolId].lastRewardedBlock = _startBlock;
204     pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
205
```

```
        emit PoolCreated("PoolCreated", currentPoolId, pools[currentPoolId],
206 block.number);
207     currentPoolId += 1;
208 }
```

Listing 31.2 The improved *createPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was acknowledged by the *Vega Investment Group* team. Nonetheless, **the team decided not to fix this issue because even the *zero* update delay is their acceptable value**.

No. 32	Malfunction Of The depositPoolReward Function		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 214 - 233 (commit id: a664de1)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: a664de1b105c3013cd7d372f48db7ea2aebeb946.

We found that the *depositPoolReward* function does not support depositing an arbitrary amount of pool reward tokens.

The function would compute the *totalPoolRewards* variable based on the pool parameters *rewardTokensPerBlock* (L219 in code snippet 32.1) and *endBlock* (L220), which are updatable parameters.

The computed *totalPoolRewards* would then be used to determine the *amount* variable (L224). Finally, the function would execute the *safeTransferFrom* function (L228) to deposit a number (specified by the computed *amount* variable) of reward tokens to the pool.

Nevertheless, we discovered that the *depositPoolReward* function does not support the following depositing scenario.

1. A staking pool is created using the following pool parameters: ***rewardTokensPerBlock* = 2, *startBlock* = 0, and *endBlock* = 10**.
2. An owner executes the *depositPoolReward* function to deposit the pool rewards. From the pool parameters described in Step 1, **the *totalPoolRewards* variable would be 22**.

   Thus, **22 reward tokens would be transferred and locked** in the *VucaStaking* contract.
3. An owner respectively invokes the functions *updateRewardTokensPerBlock* and *updateEndBlock* to **create two pool changes for updating the *rewardTokensPerBlock* parameter to 1 and the *endBlock* parameter to 15**.

4. For the sake of understanding, let's say **both pool changes are active and applied to the pool at block number 10**.

5. An owner calls the *depositPoolReward* function again to deposit additional pool rewards.

   In this step, **the computed *totalPoolRewards* variable would be 16 (i.e., *1 * (15 - 0 + 1)*), which is an incorrect value (the correct value must be 25)**.

   The incorrect *totalPoolRewards* variable (containing 16) causes the transaction to be unexpectedly reverted in L222 since the *depositPoolReward* function considers that the 22 reward tokens deposited in Step 1 are adequate for all stakers.

Even though the *depositPoolReward* function would be functioning incorrectly, an owner has a workaround solution by transferring the reward tokens to the *VucaStaking* contract directly.

With the above workaround solution, nonetheless, **the staking pool would not track the pool parameter *totalPoolRewards* (L226). That could make an owner not be able to withdraw some locked reward tokens (when calling the *retrieveReward* function) if some stakers forfeit their rewards (by calling the *emergencyWithdraw* function)**.

VucaStaking.sol

```
214 function depositPoolReward(uint16 _poolId) public {
215     Pool storage pool = pools[_poolId];
216     require(pool.inited, "Pool invalid");
217     _updatePoolInfo(_poolId);
218 
219     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
220     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
221 
222     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
223 
224     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
225 
226     pool.extension.totalPoolRewards = totalPoolRewards;
227 
228     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
229 
230     PoolChanges memory changes;
231 
232     emit PoolUpdated(2, currentPoolId, pools[_poolId], changes, block.number);
233 }
```

Listing 32.1 The *depositPoolReward* function that
does not support depositing an arbitrary amount of pool reward tokens

Recommendations

We recommend reworking the *depositPoolReward* function as per the below code snippet. The improved function would allow an owner to deposit an arbitrary amount of pool reward tokens.

VucaStaking.sol

```
214 function depositPoolReward(uint16 _poolId, uint256 _amount) public {
215     Pool storage pool = pools[_poolId];
216     require(pool.inited, "Pool invalid");
217     require(_amount > 0, "Invalid amount");
218 
219     pool.extension.totalPoolRewards += _amount;
220 
221     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    _amount);
222 
223     PoolChanges memory changes;
224 
225     emit PoolUpdated(2, _poolId, pools[_poolId], changes, block.number);
226 }
```

Listing 32.2 The improved *depositPoolReward* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue by adopting our recommended code.

No. 33	Inconsistent Error Message With The Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 247*		

Detailed Issue

We found an error message inconsistent with the code in the function *createPool* (L247 in the code snippet below). This inconsistency can lead to misunderstanding among users or developers when maintaining the source code.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");

            // (...SNIPPED...)
265 }
```

Listing 33.1 The *createPool* function with an inconsistent error message

Recommendations

We recommend revising the associated error message to reflect the actual code like L247 in the code snippet below.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid staking token");

            // (...SNIPPED...)
265 }
```

Listing 33.2 The improved *createPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue according to our suggestion.

No. 34	Inconsistent Event Emission With The Code #1		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 263*		

Detailed Issue

We found the event emission inconsistent with the operation of the *createPool* function (L263 in the code snippet below). The inconsistent event may lead to misunderstanding among developers or users when tracing the function's event log.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248 
249         pools[currentPoolId].inited = true;
250         pools[currentPoolId].rewardToken = _rewardToken;
251         pools[currentPoolId].stakeToken = _stakeToken;
252 
253         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
254         pools[currentPoolId].startBlock = _startBlock;
255         pools[currentPoolId].endBlock = _endBlock;
256 
257         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
258         pools[currentPoolId].lastRewardedBlock = _startBlock;
259         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
```

```
260
261         PoolChanges memory changes;
262
263         emit PoolUpdated(currentPoolId, pools[currentPoolId], changes,
    block.number);
264         currentPoolId += 1;
265 }
```

Listing 34.1 The inconsistent event emission in the *createPool* function

Recommendations

We recommend **emitting the new event to be consistent with the *createPool* function** as shown in L263 in the code snippet below.

VucaStaking.sol

```
236 function createPool(
237         address _rewardToken,
238         address _stakeToken,
239         uint256 _maxStakeTokens,
240         uint256 _startBlock,
241         uint256 _endBlock,
242         uint256 _rewardTokensPerBlock,
243         uint32 _updateDelay
244     ) external onlyOwner {
245         require(_startBlock > 0 && _startBlock < _endBlock, "Invalid start/end
    block");
246         require(_rewardToken != address(0), "Invalid reward token");
247         require(_stakeToken != address(0), "Invalid reward token");
248
249         pools[currentPoolId].inited = true;
250         pools[currentPoolId].rewardToken = _rewardToken;
251         pools[currentPoolId].stakeToken = _stakeToken;
252
253         pools[currentPoolId].maxStakeTokens = _maxStakeTokens;
254         pools[currentPoolId].startBlock = _startBlock;
255         pools[currentPoolId].endBlock = _endBlock;
256
257         pools[currentPoolId].rewardTokensPerBlock = _rewardTokensPerBlock *
    (10**IERC20(_stakeToken).decimals()) * REWARDS_PRECISION;
258         pools[currentPoolId].lastRewardedBlock = _startBlock;
259         pools[currentPoolId].updateDelay = _updateDelay; // = 8 hours;
260
261         PoolChanges memory changes;
262
263         emit PoolCreated(currentPoolId, pools[currentPoolId], block.number);
264         currentPoolId += 1;
```

```
265 }
```

Listing 34.2 The improved event in the *createPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue according to our recommendation.

No. 35	Recommended Enforcing Checks-Effects-Interactions Pattern		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 153 - 176 and 179 - 208*		

Detailed Issue

We noticed that the functions ***emergencyWithdraw*** (code snippet 35.1) and ***unStake*** (code snippet 35.2) **do not follow the *checks-effects-interactions* pattern**, which is the best practice coding style to prevent potential *reentrancy* attacks.

In L169 in the code snippet 35.1 below, for example, the *emergencyWithdraw* function transfers a staking token back to a staker (*interactions part*) before updating state variables (*effects part*) in L171 - 175.

Even if there are no *reentrancy* issues, we recommend that both functions should be enforced the *checks-effects-interactions* pattern.

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
154     _updatePoolInfo(_poolId);
155     Pool storage pool = pools[_poolId];
156     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
157     uint256 amount = staking.amount;
158     require(staking.amount > 0, "Insufficient funds");
159 
160     _updatePoolRewards(_poolId, block.number);
161     // Update pool
162     if (pool.tokensStaked >= amount) {
163         pool.tokensStaked -= amount;
164     }
165 
166     staking.amount = 0;
167 
168     // Withdraw tokens
169     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
170 
171     emit StakingChanged(msg.sender, _poolId, pool, staking);
```

```
172
173     // Update staker
174     staking.accumulatedRewards = 0;
175     staking.minusRewards = 0;
176 }
```

Listing 35.1 The *emergencyWithdraw* function
without enforcing the *checks-effects-interactions* pattern

VucaStaking.sol

```
179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock <= block.number, "Staking active");
183
184     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
185     uint256 amount = staking.amount;
186     require(staking.amount > 0, "Insufficient funds");
187
188     _updatePoolRewards(_poolId, block.number);
189     // Pay rewards
190     uint256 rewards = getRewards(_poolId, msg.sender);
191     IERC20(pool.rewardToken).transfer(msg.sender, rewards);
192
193     // Update pool
194     pool.rewardsWithdrew += getRawRewards(_poolId, msg.sender);
195     if (pool.tokensStaked >= amount) {
196         pool.tokensStaked -= amount;
197     }
198
199     // Withdraw tokens
200     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
201
202     emit StakingChanged(msg.sender, _poolId, pool, staking);
203
204     // Update staker
205     staking.accumulatedRewards = 0;
206     staking.minusRewards = 0;
207     staking.amount = 0;
208 }
```

Listing 35.2 The *unStake* function
without enforcing the *checks-effects-interactions* pattern

Recommendations

We recommend enforcing the *checks-effects-interactions* pattern to both ***emergencyWithdraw*** (code snippet 35.3) and ***unStake*** (code snippet 35.4) functions.

To fix this issue in detail, we moved the *interactions* part (the *transfer* function in L175 in the below code snippet 35.3) to get executed after the *effects* part (L168 - 172).

VucaStaking.sol

```
153 function emergencyWithdraw(uint16 _poolId) external {
154     _updatePoolInfo(_poolId);
155     Pool storage pool = pools[_poolId];
156     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
157     uint256 amount = staking.amount;
158     require(staking.amount > 0, "Insufficient funds");
159 
160     _updatePoolRewards(_poolId, block.number);
161     // Update pool
162     if (pool.tokensStaked >= amount) {
163         pool.tokensStaked -= amount;
164     }
165 
166     staking.amount = 0;
167 
168     emit StakingChanged(msg.sender, _poolId, pool, staking);
169 
170     // Update staker
171     staking.accumulatedRewards = 0;
172     staking.minusRewards = 0;
173 
174     // Withdraw tokens
175     IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
176 }
```

Listing 35.3 The improved *emergencyWithdraw* function enforcing the *checks-effects-interactions* pattern

VucaStaking.sol

```
179 function unStake(uint16 _poolId) external {
180     _updatePoolInfo(_poolId);
181     Pool storage pool = pools[_poolId];
182     require(pool.endBlock <= block.number, "Staking active");
183 
184     Staking storage staking = stakingUsersInfo[_poolId][msg.sender];
185     uint256 amount = staking.amount;
186     require(staking.amount > 0, "Insufficient funds");
187 
```

```
188         _updatePoolRewards(_poolId, block.number);
189         uint256 rewards = getRewards(_poolId, msg.sender);
190 
191         // Update pool
192         pool.rewardsWithdrew += getRawRewards(_poolId, msg.sender);
193         if (pool.tokensStaked >= amount) {
194             pool.tokensStaked -= amount;
195         }
196 
197         emit StakingChanged(msg.sender, _poolId, pool, staking);
198 
199         // Update staker
200         staking.accumulatedRewards = 0;
201         staking.minusRewards = 0;
202         staking.amount = 0;
203 
204         // Pay rewards
205         IERC20(pool.rewardToken).transfer(msg.sender, rewards);
206 
207         // Withdraw tokens
208         IERC20(pool.stakeToken).transfer(address(msg.sender), amount);
209     }
```

Listing 35.4 The improved *unStake* function
enforcing the *checks-effects-interactions* pattern

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team fixed this issue by enforcing the *checks-effects-interactions* pattern.

No. 36	Inconsistent Event Emission With The Code #2		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/VucaStaking.sol*		
Locations	*VucaStaking.sol L: 209 - 227 (commit id: 5cc2e3f)*		

Detailed Issue

This issue was raised during the reassessment phase at the commit id: 5cc2e3fcb2a4268bd97e6e02395bac08b592a91d.

We found an event emission inconsistent with the operation of the *depositPoolReward* function (L226 in code snippet 36.1). The inconsistent event may lead to misunderstanding among developers or users when tracing the function's event log.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
(10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
amount);
221 
222     pool.extension.totalPoolRewards = totalPoolRewards;
223 
224     PoolChanges memory changes;
225 
226     emit PoolUpdated("PoolUpdated", currentPoolId, pools[_poolId], changes,
block.number);
```

```
227 }
```

Listing 36.1 The inconsistent event emission in the *depositPoolReward* function

Recommendations

We recommend **emitting the new relevant event to be consistent with the *depositPoolReward* function** as shown in L224 in the code snippet below.

VucaStaking.sol

```
209 function depositPoolReward(uint16 _poolId) public {
210     Pool storage pool = pools[_poolId];
211     require(pool.inited, "Pool invalid");
212 
213     uint256 rewardTokenPerBlock = pool.rewardTokensPerBlock /
    (10**IERC20Helper(pool.stakeToken).decimals()) / REWARDS_PRECISION;
214     uint256 totalPoolRewards = rewardTokenPerBlock * (pool.endBlock -
    pool.startBlock + 1);
215 
216     require(totalPoolRewards > pool.extension.totalPoolRewards, "Already
    deposited");
217 
218     uint256 amount = totalPoolRewards - pool.extension.totalPoolRewards;
219 
220     IERC20(pool.rewardToken).safeTransferFrom(msg.sender, address(this),
    amount);
221 
222     pool.extension.totalPoolRewards = totalPoolRewards;
223 
224     emit PoolRewardDeposited(_poolId, amount, totalPoolRewards, block.number);
225 }
```

Listing 36.2 The improved event in the *depositPoolReward* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Vega Investment Group* team acknowledged this issue but decided not to fix it because **the current implementation emits event parameters expected by their off-chain web services**.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information


info@valix.io


https://www.facebook.com/ValixConsulting


https://twitter.com/ValixConsulting


https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/