

Staking Pool

Smart Contract Audit Report Prepared for deFusion

[deFusion]

Date Issued:	Mar 11, 2024
Project ID:	AUDIT2023021
Version:	v1.0
Confidentiality Level:	Public



Report Information

Project ID	AUDIT2023021
Version	v1.0
Client	deFusion
Project	Staking Pool
Auditor(s)	Natsasit Jirathammanuwat Kongkit Chatchawanhirun
Author(s)	Natsasit Jirathammanuwat
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Mar 11, 2024	Full report	Natsasit Jirathammanuwat

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary	1
1.1. Audit Result	1
1.2. Disclaimer	1
2. Project Overview	2
2.1. Project Introduction	2
2.2. Scope	3
2.3. Security Model	5
3. Methodology	6
3.1. Test Categories	6
3.2. Audit Items	7
3.3. Risk Rating	9
4. Summary of Findings	10
5. Detailed Findings Information	12
5.1. Improper Withdrawal State Handling When Unstake and Resign in the Same Block	12
5.2. Lack of lastBalance state Update	19
5.3. Incorrect Withdrawal State Check in the withdrawAfterResign() Function	22
5.4. Lack of Zero Withdrawal Handling	26
5.5. Incorrect Period Validation in the withdrawFromValidator() Function	31
5.6. Centralized Authority Control	34
5.7. Improper Input Validation	36
5.8. Insufficient Check in the propose() Function	40
5.9. Outdated Compiler Version	43
5.10. Incorrect Event Logging	45
5.11. Inexplicit Solidity Compiler Version	47
5.12. Improper Function Visibility	48
6. Appendix	50
6.1. About Inspex	50

1. Executive Summary

As requested by deFusion, Inspex team conducted an audit to verify the security posture of the Staking Pool smart contracts between Dec 13, 2023 and Dec 14, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Staking Pool smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 critical, 3 high, 2 medium, 1 low, 2 very low, and 3 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved in the reassessment. Therefore, Inspex trusts that Staking Pool smart contracts have high-level protections in place to be safe from most attacks.

However, as the source code is currently not publicly available, there is a potential risk that the smart contracts deployed on the blockchain may not be identical to the audited smart contracts. This discrepancy could result in introducing security vulnerabilities or unintended behaviors that were not identified during the audit process. It is of importance to recognize that interacting with an unverified smart contract may lead to the potential loss of funds. In this case, the hash of the deployed smart contract bytecode should be compared with the hash of the audited smart contract bytecode to ensure that the deployed smart contract is identical to the audited smart contract before interacting with them. In the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

deFusion, where trust meets power. The deFusion staking pool is designed exclusively for whales seeking a secure and reliable B2B staking experience. With cutting-edge technology and a commitment to transparency, deFusion provides a trustworthy platform for users to stake their tokens, ensuring a fusion of stability and strength in the ever-evolving blockchain landscape.

Scope Information:

Project Name	Staking Pool
Website	https://www.defusion.xyz/
Smart Contract Type	Ethereum Smart Contract
Chain	Viction
Programming Language	Solidity
Category	Staking Pool

Audit Information:

Audit Method	Whitebox
Audit Date	Dec 13, 2023 - Dec 14, 2023
Reassessment Date	Jan 2, 2024

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox:** The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox:** Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The smart contracts with the following bytecodes were audited and reassessed by Inspex in detail:

Initial Audit:

Contract	Bytecode SHA256 Hash
Vfactory	c3cde9a4ce01a385f83f12c4a29e74ca9f923a25155e9844093a1495f3890be2
Vpool	d8f306b78c9d13782bba36e7608b1820412f6e288c9ab50e1d2948775ced7099
libraries/Context	-
libraries/Ownable	-
libraries/Payable	-
libraries/ProxyFactory	-
libraries/SafeMath	-

hardhat.config.ts

```
{
  solidity: {
    version: "0.8.16",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  }
}
```

Reassessment Audit:

Contract	Bytecode SHA256 Hash
DefusionFactory	ad4fba2ad287ab938f13446269f7af62a7adfd4b25606a48613b7a1b2b6673e5
Vpool	c12717040eae86fb76e5e7e935714e8b4a76f178fd19edcde412d02429c1b11c
libraries/Context	-
libraries/Ownable	-
libraries/Payable	-
libraries/ProxyFactory	-

libraries/SafeMath

-

hardhat.config.ts

```
{
  solidity: {
    version: "0.8.19",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  }
}
```

As the deFusion team has decided not to publish the source code to protect their intellectual property, the users should compare the bytecode hashes with the smart contracts deployed before interacting with them to make sure that they are the same with the contracts audited.

2.3. Security Model

2.3.1 Trust Modules

The deFusion has a privileged role with the authority to mutate the critical state variables of the contract. Changes to these state variables significantly impact the contract's functionality. The privileged role and their corresponding privileged function are enumerated as follows:

- The owner address can set the pool implementation by executing the **setImplement()** function. Changing the pool implementation could possibly result in unauthorized minting or burning of the **Vfactory** tokens; this can be maliciously used to permanently lock users' staked funds.

The deFusion several functionalities have relied on the external component, which may significantly impact the contract if they malfunction. The external components are listed as follows:

- The **TomoValidator** contract offers a range of functionalities related to the validator staking reward, spanning from proposing and voting to unvoting, resigning, and withdrawing.

2.3.2 Trust Assumptions

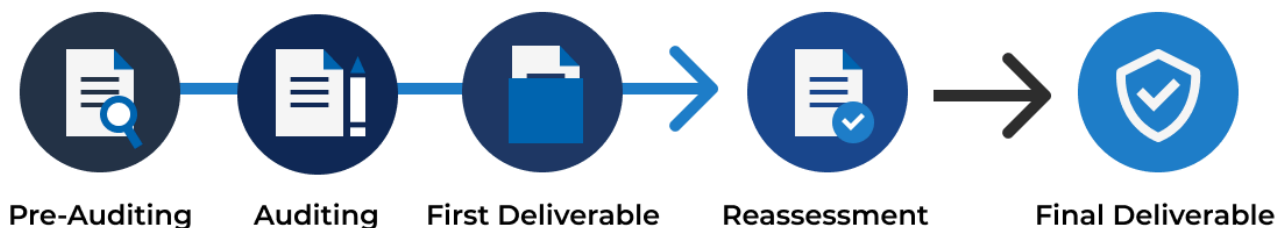
In the deFusion, the protocol's privileged role, has the ability to change the critical state variable of the contract, also external components such as the **TomoValidator** contract were assumed to be trusted. Acknowledging these trust assumptions is important, as it introduces substantial risks to the platform. Trust assumptions include, but are not limited to:

- All privileged addresses perform the privileged function with good will. The owner address is trusted to change the pool contract implementation to only legitimate contracts.
- It is presumed that the **TomoValidator** contract consistently functions as intended.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing:** Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing:** Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting:** Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment:** Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable:** Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at (<https://docs.inspex.co/smart-contract-security-testing-guide/>).

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	<ul style="list-style-type: none">1.1. Proper measures should be used to control the modifications of smart contract logic1.2. The latest stable compiler version should be used1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds1.4. The smart contract source code should be publicly available1.5. State variables should not be unfairly controlled by privileged accounts1.6. Least privilege principle should be used for the rights of each role
2. Access Control	<ul style="list-style-type: none">2.1. Contract self-destruct should not be done by unauthorized actors2.2. Contract ownership should not be modifiable by unauthorized actors2.3. Access control should be defined and enforced for each actor roles2.4. Authentication measures must be able to correctly identify the user2.5. Smart contract initialization should be done only once by an authorized party2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	<ul style="list-style-type: none">3.1. Function return values should be checked to handle different results3.2. Privileged functions or modifications of critical states should be logged3.3. Modifier should not skip function execution without reverting
4. Business Logic	<ul style="list-style-type: none">4.1. The business logic implementation should correspond to the business design4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions4.3. msg.value should not be used in loop iteration
5. Blockchain Data	<ul style="list-style-type: none">5.1. Result from random value generation should not be predictable5.2. Spot price should not be used as a data source for price oracles5.3. Timestamp should not be used to execute critical functions5.4. Plain sensitive data should not be stored on-chain5.5. Modification of array state should not be done by value5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	<ul style="list-style-type: none">6.1. Unknown external components should not be invoked6.2. Funds should not be approved or transferred to unknown accounts6.3. Reentrant calling should not negatively affect the contract states6.4. Vulnerable or outdated components should not be used in the smart contract6.5. Deprecated components that have no longer been supported should not be used in the smart contract6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	<ul style="list-style-type: none">7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows7.2. Explicit conversion of types should be checked to prevent unexpected results7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	<ul style="list-style-type: none">8.1. State changing functions that loop over unbounded data structures should not be used8.2. Unexpected revert should not make the whole smart contract unusable8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	<ul style="list-style-type: none">9.1. State and function visibility should be explicitly labeled9.2. Token implementation should comply with the standard specification9.3. Floating pragma version should not be used9.4. Builtin symbols should not be shadowed9.5. Functions that are never called internally should not have public visibility9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood:** a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact:** a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

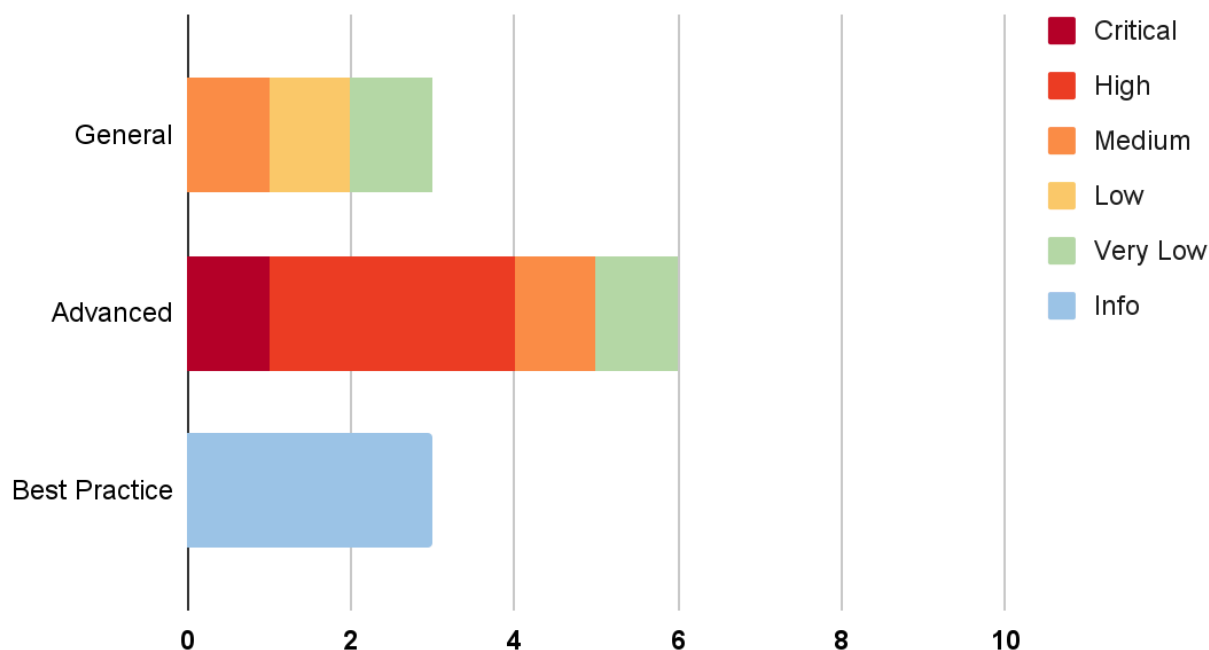
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Likelihood		
	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

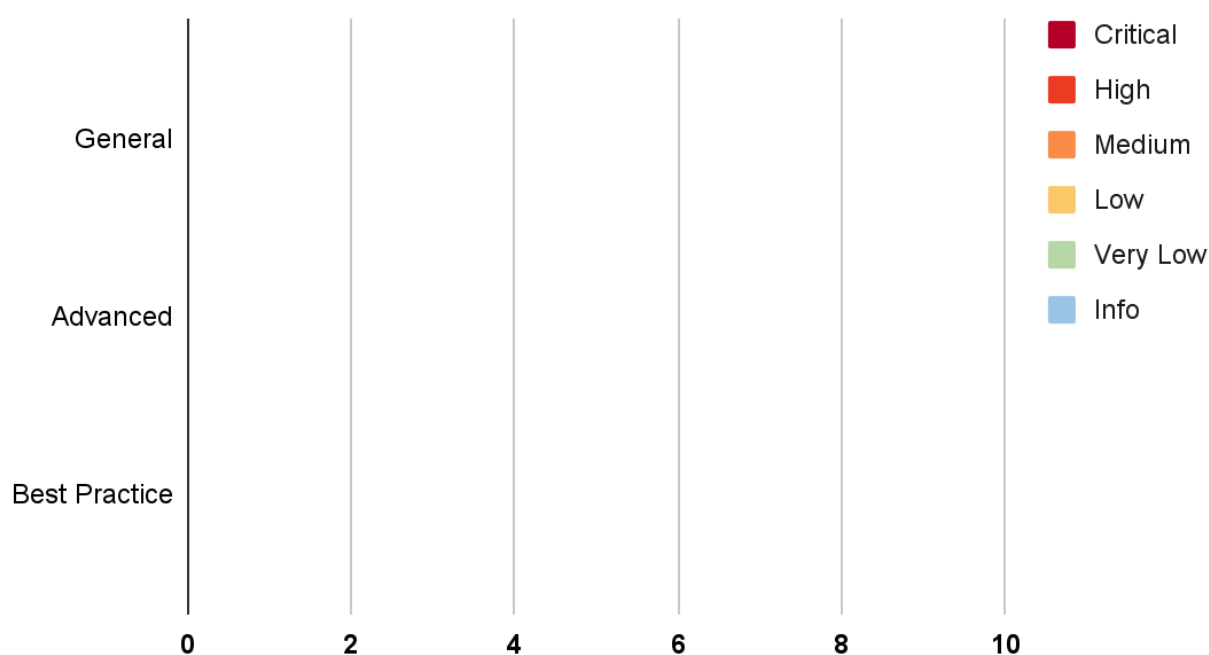
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Improper Withdrawal State Handling When Unstake and Resign in the Same Block	Advanced	Critical	Resolved
IDX-002	Lack of lastBalance state Update	Advanced	High	Resolved
IDX-003	Incorrect Withdrawal State Check in the withdrawAfterResign() Function	Advanced	High	Resolved
IDX-004	Lack of Zero Withdrawal Handling	Advanced	High	Resolved
IDX-005	Incorrect Period Validation in the withdrawFromValidator() Function	Advanced	Medium	Resolved
IDX-006	Centralized Authority Control	General	Medium	Resolved
IDX-007	Improper Input Validation	General	Low	Resolved
IDX-008	Insufficient Check in the propose() Function	Advanced	Very Low	Resolved
IDX-009	Outdated Compiler Version	General	Very Low	Resolved
IDX-010	Incorrect Event Logging	Best Practice	Info	Resolved
IDX-011	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-012	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by deFusion can be found in Chapter 5.

5. Detailed Findings Information

5.1. Improper Withdrawal State Handling When Unstake and Resign in the Same Block

ID	IDX-001
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Critical</p> <p>Impact: High The user withdrawal fund will be considered as a platform reward. The attacker can abuse this vulnerability and drain users' funds as a reward and permanently lock users' funds.</p> <p>Likelihood: High This vulnerability arises when the <code>unstake()</code> and <code>resign()</code> functions are executed in the same block. This issue can occur unintentionally by the user and may also be intentionally triggered by an attacker.</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by adding the validation withdrawal block number already created in the <code>resign()</code> function.</p>

5.1.1. Description

Normally, while the `poolStatus == PoolStatus.PROPOSED`, the user can call the `unstake()` function to initiate the withdrawal process. This process will set the `withdrawIndexes[withdrawBlockNumber]` state at line 199 and call `unvote()` to the `TomoValidator` contract as shown in line 205.

Pool.sol

```

171 function unstake(uint256 _amount) external nonReentrant {
172     UserInfo storage user = users[msg.sender];
173     require(user.capacity >= _amount, "vPool: Insufficient amount to
withdraw");
174     require(poolStatus != PoolStatus.RESIGNED, "vPool: Pool is resigned");
175
176     updatePool();
177     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
178
179     if (pending > 0) {
180         payable(msg.sender).transfer(pending);

```

```

181         emit WithdrawReward(msg.sender, pending);
182     }
183
184     if (poolStatus == PoolStatus.PROPOSED) {
185         lastBalance = address(this).balance;
186     }
187
188     user.capacity = user.capacity.sub(_amount);
189     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
190
191     if (_amount > 0 && poolStatus == PoolStatus.PENDING) {
192         factory.burnLpToken(msg.sender, _amount);
193         payable(msg.sender).transfer(_amount);
194     }
195
196     if (_amount > 0 && poolStatus == PoolStatus.PROPOSED) {
197         uint256 withdrawBlockNumber = unvoteDelayedBlocks.add(block.number);
198         require(totalCapacity.sub(_amount) >= 50000 ether, "vPool: Unstake
amount too large");
199         require(withdrawIndexes[withdrawBlockNumber] == 0, "vPool: Withdraw
block number already created, please wait for another block");
200
201         withdrawsState[msg.sender].caps[withdrawBlockNumber] =
withdrawsState[msg.sender].caps[withdrawBlockNumber].add(_amount);
202         withdrawsState[msg.sender].blockNumbers.push(withdrawBlockNumber);
203         withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
204         currentWithdrawIndex = currentWithdrawIndex.add(1);
205         tomoValidator.unvote(coinbase, _amount);
206
207         if (votingManagement.voteResults[msg.sender]) {
208             votingManagement.totalSupportCap =
votingManagement.totalSupportCap.sub(_amount);
209             emit CommunityVote(msg.sender, _amount, false);
210         }
211     }
212
213     totalCapacity = totalCapacity.sub(_amount);
214     emit Unstake(msg.sender, _amount);
215 }

```

In the TomoValidator contract, the unvoted amount will be accumulated in the withdrawsState[msg.sender].caps[withdrawBlockNumber] state at line 218.

Validator.sol

```

212 function unvote(address _candidate, uint256 _cap) public
onlyValidVote(_candidate, _cap) {
213     validatorsState[_candidate].cap =

```

```

validatorsState[_candidate].cap.sub(_cap);
214     validatorsState[_candidate].voters[msg.sender] =
validatorsState[_candidate].voters[msg.sender].sub(_cap);
215
216     // refund after delay X blocks
217     uint256 withdrawBlockNumber = voterWithdrawDelay.add(block.number);
218     withdrawsState[msg.sender].caps[withdrawBlockNumber] =
withdrawsState[msg.sender].caps[withdrawBlockNumber].add(_cap);
219     withdrawsState[msg.sender].blockNumbers.push(withdrawBlockNumber);
220
221     emit Unvote(msg.sender, _candidate, _cap);
222 }

```

Which can be withdrawn later by executing `withdraw()` function. The native token will be transferred in the number of the stored `withdrawsState[msg.sender].caps[_blockNumber]` amount state.

Validator.sol

```

243 function withdraw(uint256 _blockNumber, uint _index) public
onlyValidWithdraw(_blockNumber, _index) {
244     uint256 cap = withdrawsState[msg.sender].caps[_blockNumber];
245     delete withdrawsState[msg.sender].caps[_blockNumber];
246     delete withdrawsState[msg.sender].blockNumbers[_index];
247     msg.sender.transfer(cap);
248     emit Withdraw(msg.sender, _blockNumber, cap);
249 }

```

However, the `resign()` function in the `Vpool` contract also contains an `unvote()` function call to the `TomoValidator` contract at line 346.

Pool.sol

```

336 function resign() public canResign {
337     require(poolStatus == PoolStatus.PROPOSED, "vPool: pool is not proposed");
338
339     updatePool();
340     lastBalance = address(this).balance;
341
342     afterResignState.capBeforeResign = totalCapacity;
343     uint256 _unvoteAmount = totalCapacity.sub(50000 ether);
344
345     // Unvote before resigning prevents the need to wait for 30 days to
withdraw
346     tomoValidator.unvote(coinbase, _unvoteAmount);
347
348     uint256 withdrawBlockNumber = unvoteDelayedBlocks.add(block.number);
349     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(_unvoteAmount);

```

```

350     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
351     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
352     currentWithdrawIndex = currentWithdrawIndex.add(1);
353
354     tomoValidator.resign(coinbase);
355
356     withdrawBlockNumber = resignDelayedBlocks.add(block.number);
357     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(
358         totalCapacity.sub(_unvoteAmount)
359     );
360     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
361     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
362     currentWithdrawIndex = currentWithdrawIndex.add(1);
363
364     poolStatus = PoolStatus.RESIGNED;
365     resignBlock = block.number;
366
367     emit Resign(address(this));
368 }

```

With this implementation, there is no prevention to check whether the `withdrawIndexes[withdrawBlockNumber]` is already assigned by the `unstake()` function called by the user.

If the `unstake()` and the `resign()` functions are called in the same block, the `withdrawsState[msg.sender].caps[_blockNumber]` state of the `TomoValidator` contract will be accumulated by both the user's unstaked fund and the platform users' unvoted fund.

The attacker can exploit this issue by executing the `unstake()` function and `resign()` in the same block. Then call the `withdrawUnstakedAmount()` function to withdraw the accumulated fund (attacker's unstaked fund + platform's users fund). The unstaked fund will be transferred to the attacker as shown in line 236, and the platform's users fund will remain in the `Vpool` contract.

Pool.sol

```

219 function withdrawUnstakedAmount(uint256 _blockNumber) public {
220     require(_blockNumber > 0, "vPool: Withdraw block number must be greater
than 0");
221     require(poolStatus != PoolStatus.PENDING, "vPool: Pool is pending");
222
223     uint256 cap = withdrawsState[msg.sender].caps[_blockNumber];
224     require(block.number >= _blockNumber, "vPool: Withdraw block number must be
less than current block number");
225     require(cap > 0, "vPool: No withdraw cap");
226     require(withdrawIndexes[_blockNumber] > 0, "vPool: No withdraw index");
227

```

```

228     uint256 balanceBefore = address(this).balance;
229     tomoValidator.withdraw(_blockNumber, withdrawIndexes[_blockNumber].sub(1));
230
231     uint256 balanceAfter = address(this).balance;
232     require(balanceAfter.sub(balanceBefore) >= cap);
233     delete withdrawsState[msg.sender].caps[_blockNumber];
234     delete withdrawIndexes[_blockNumber];
235     factory.burnLpToken(msg.sender, cap);
236     payable(msg.sender).transfer(cap);
237     emit Withdraw(msg.sender, _blockNumber, cap);
238 }

```

Finally, the attacker can call the `withdrawReward()` function, which considers the balance in the contract as a reward and distributes it to the attacker.

Pool.sol

```

298 function withdrawReward() public nonReentrant {
299     UserInfo storage user = users[msg.sender];
300
301     updatePool();
302     uint256 pending =
303     user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
304
305     if (pending > 0) {
306         payable(msg.sender).transfer(pending);
307         emit WithdrawReward(msg.sender, pending);
308     }
309
310     if (poolStatus == PoolStatus.PROPOSED || (poolStatus == PoolStatus.RESIGNED
311     && !afterResignState.isWithdrawUnstakeLock)) {
312         lastBalance = address(this).balance;
313     }
314
315     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
316 }

```

This also reverts the `withdrawFromValidator()` function execution at line 285 due to the `withdrawIndexes[_blockNumber]` state being deleted in the `withdrawUnstakedAmount()` at line 234. Resulting in a loss of users' funds.

Pool.sol

```

281 function withdrawFromValidator() public {
282     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&
283     !afterResignState.isWithdrawUnstakeLock) {
284         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];

```

```

285         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
286         afterResignState.isWithdrawUnstakeLock = true;
287     }
288
289     if (block.number > resignBlock.add(resignDelayedBlocks) &&
!afterResignState.isWithdrawResignLock) {
290         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];
291
292         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
293         afterResignState.isWithdrawResignLock = true;
294     }
295 }

```

5.1.2. Remediation

Inspex suggests preventing the `resign()` function execution in the same block with the `unstake()` function by adding the validation at lines 346 - 347:

Pool.sol

```

336 function resign() public canResign {
337     require(poolStatus == PoolStatus.PROPOSED, "vPool: pool is not proposed");
338
339     updatePool();
340     lastBalance = address(this).balance;
341
342     afterResignState.capBeforeResign = totalCapacity;
343     uint256 _unvoteAmount = totalCapacity.sub(50000 ether);
344
345     // Unvote before resigning prevents the need to wait for 30 days to
withdraw
346     uint256 withdrawBlockNumber = unvoteDelayedBlocks.add(block.number);
347     require(withdrawIndexes[withdrawBlockNumber] == 0, "vPool: Withdraw block
number already created, please wait for another block");
348
349     tomoValidator.unvote(coinbase, _unvoteAmount);
350
351     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(_unvoteAmount);
352     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
353     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
354     currentWithdrawIndex = currentWithdrawIndex.add(1);
355
356     tomoValidator.resign(coinbase);
357
358     withdrawBlockNumber = resignDelayedBlocks.add(block.number);

```

```
359     withdrawsState[address(this)].caps[withdrawBlockNumber] =  
withdrawsState[address(this)].caps[withdrawBlockNumber].add(  
360         totalCapacity.sub(_unvoteAmount)  
361     );  
362     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);  
363     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;  
364     currentWithdrawIndex = currentWithdrawIndex.add(1);  
365  
366     poolStatus = PoolStatus.RESIGNED;  
367     resignBlock = block.number;  
368  
369     emit Resign(address(this));  
370 }
```

5.2. Lack of lastBalance state Update

ID	IDX-002
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: High</p> <p>Impact: High The balance in the contract will be calculated as a reward multiple times, which results in an inflation of the reward.</p> <p>Likelihood: Medium This issue will occur when the <code>updatePool()</code> function is executed and the <code>accRewardPerShare</code> state is updated without a <code>lastBalance</code> state update afterward.</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by adding the <code>lastBalance</code> state update in the <code>withdrawAfterResign()</code> function.</p>

5.2.1. Description

Normally, the `updatePool()` function is called in various functions in order to update the `accRewardPerShare` state with the current reward (get from the `currentReward()` function).

Pool.sol

```

124 function updatePool() private {
125     if (poolStatus == PoolStatus.PROPOSED || (poolStatus == PoolStatus.RESIGNED
&& !afterResignState.isWithdrawUnstakeLock)) {
126         uint256 reward = currentReward();
127         accRewardPerShare =
accRewardPerShare.add(reward.mul(PRECISION).div(totalCapacity));
128     }
129 }
```

The current reward is calculated from the current contract balance deducted by the `lastBalance` state as shown in line 88.

Pool.sol

```

86 function currentReward() private view returns (uint256) {
87     if (poolStatus == PoolStatus.PROPOSED || (poolStatus == PoolStatus.RESIGNED
&& !afterResignState.isWithdrawUnstakeLock)) {
88         return address(this).balance.sub(lastBalance);

```

```
89     }
90     return 0;
91 }
```

If the `updatePool()` function is called but never updates the `lastBalance` state afterward, it will result in the reward inflation.

The following `withdrawAfterResign()` function calls the `updatePool()` function at line 247, but never updates the `lastBalance` state. Repeatedly executing the `withdrawAfterResign()` function in some condition will inflate the reward.

5.2.2. Remediation

Inspex suggests implementing the `lastBalance` state update in the `withdrawAfterResign()` function as shown in lines 275 - 277:

Pool.sol

```
241 function withdrawAfterResign() public nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245     uint256 stakerCap = user.capacity;
246
247     updatePool();
248     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
249
250     if (pending > 0) {
251         payable(msg.sender).transfer(pending);
252         emit WithdrawReward(msg.sender, pending);
253     }
254
255     withdrawFromValidator();
256
257     if (
258         afterResignState.isWithdrawUnstakeLock &&
!afterResignState.isWithdrawResignLock &&
!afterResignState.userWithdrawnAfterResign[msg.sender]
259     ) {
260         stakerCap = afterResignState.capBeforeResign.sub(50000
ether).mul(stakerCap).div(afterResignState.capBeforeResign);
261         user.capacity = user.capacity.sub(stakerCap);
262         totalCapacity = totalCapacity.sub(stakerCap);
263         afterResignState.userWithdrawnAfterResign[msg.sender] = true;
264
265         factory.burnLpToken(msg.sender, stakerCap);
```

```
266     payable(msg.sender).transfer(stakerCap);
267 } else {
268     factory.burnLpToken(msg.sender, stakerCap);
269     totalCapacity = totalCapacity.sub(stakerCap);
270
271     user.capacity = 0;
272     payable(msg.sender).transfer(stakerCap);
273 }
274
275 if (!afterResignState.isWithdrawUnstakeLock) {
276     lastBalance = address(this).balance;
277 }
278 user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
279
280 emit WithdrawStakeAfterResign(msg.sender, stakerCap);
281 }
```

5.3. Incorrect Withdrawal State Check in the `withdrawAfterResign()` Function

ID	IDX-003
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: High</p> <p>Impact: Medium Users might be able to fully withdraw after two days without waiting for 30 days, and some may not be able to partially withdraw their unstaked assets or rewards until the full 30-day waiting period after resigning.</p> <p>Likelihood: High It is very likely that users will call this function to fully withdraw their assets without waiting for 30 days.</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by implementing the <code>afterResignState.isWithdrawResignLock</code> state validation in the <code>withdrawAfterResign()</code> function as suggested.</p>

5.3.1. Description

The `withdrawAfterResign()` function allows users to withdraw their staked assets and any accumulated rewards after the pool has been resigned, subject to a two-day waiting period. It includes additional considerations for the unstaked lock period.

In lines 257-266, these lines enable users to partially withdraw their unstaked assets after resigning, subject to a 2-day waiting period which is the unvoted amount after resignation.

In lines 267-273, these lines enable users to fully withdraw their unstaked assets after resigning, subject to a 30-day waiting period which is a locked amount after resignation.

Pool.sol

```

241 function withdrawAfterResign() public nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245     uint256 stakerCap = user.capacity;
246
247     updatePool();

```

```

248     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
249
250     if (pending > 0) {
251         payable(msg.sender).transfer(pending);
252         emit WithdrawReward(msg.sender, pending);
253     }
254
255     withdrawFromValidator();
256
257     if (
258         afterResignState.isWithdrawUnstakeLock &&
!afterResignState.isWithdrawResignLock &&
!afterResignState.userWithdrawnAfterResign[msg.sender]
259     ) {
260         stakerCap = afterResignState.capBeforeResign.sub(50000
ether).mul(stakerCap).div(afterResignState.capBeforeResign);
261         user.capacity = user.capacity.sub(stakerCap);
262         totalCapacity = totalCapacity.sub(stakerCap);
263         afterResignState.userWithdrawnAfterResign[msg.sender] = true;
264
265         factory.burnLpToken(msg.sender, stakerCap);
266         payable(msg.sender).transfer(stakerCap);
267     } else {
268         factory.burnLpToken(msg.sender, stakerCap);
269         totalCapacity = totalCapacity.sub(stakerCap);
270
271         user.capacity = 0;
272         payable(msg.sender).transfer(stakerCap);
273     }
274
275     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
276
277     emit WithdrawStakeAfterResign(msg.sender, stakerCap);
278 }

```

However, users can fully withdraw after two days without waiting for 30 days by calling the `withdrawAfterResign()` function two times.

At line 258, the conditional check determines whether the user has already partially withdrawn their assets. If this condition is met, it sets `afterResignState.userWithdrawnAfterResign[msg.sender]` to `true` at line 263.

Pool.sol

```

257     if (
258         afterResignState.isWithdrawUnstakeLock &&

```

```

!afterResignState.isWithdrawResignLock &&
!afterResignState.userWithdrawnAfterResign[msg.sender]
259     ) {
260         stakerCap = afterResignState.capBeforeResign.sub(50000
ether).mul(stakerCap).div(afterResignState.capBeforeResign);
261         user.capacity = user.capacity.sub(stakerCap);
262         totalCapacity = totalCapacity.sub(stakerCap);
263         afterResignState.userWithdrawnAfterResign[msg.sender] = true;
264
265         factory.burnLpToken(msg.sender, stakerCap);
266         payable(msg.sender).transfer(stakerCap);
267     } else {
268         factory.burnLpToken(msg.sender, stakerCap);
269         totalCapacity = totalCapacity.sub(stakerCap);
270
271         user.capacity = 0;
272         payable(msg.sender).transfer(stakerCap);
273     }

```

As a result, since `afterResignState.userWithdrawnAfterResign[msg.sender]` is now `true`, the user can call this function again to enter the else condition and fully withdraw unstaked assets, which may include other users' unstaked assets or rewards.

In such a scenario, certain users may encounter the inability to withdraw their partially unstaked assets until the 30-day resignation delay has passed.

5.3.2. Remediation

Inspex suggests ensuring the full withdrawal is available by checking if the `afterResignState.isWithdrawResignLock` is `true` at line 268. For example:

Pool.sol

```

241 function withdrawAfterResign() public nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245     uint256 stakerCap = user.capacity;
246
247     updatePool();
248     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
249
250     if (pending > 0) {
251         payable(msg.sender).transfer(pending);
252         emit WithdrawReward(msg.sender, pending);
253     }

```

```
254     withdrawFromValidator();
255
256     if (
257         afterResignState.isWithdrawUnstakeLock &&
258         !afterResignState.isWithdrawResignLock &&
259         !afterResignState.userWithdrawnAfterResign[msg.sender]
260     ) {
261         stakerCap = afterResignState.capBeforeResign.sub(50000
262 ether).mul(stakerCap).div(afterResignState.capBeforeResign);
263         user.capacity = user.capacity.sub(stakerCap);
264         totalCapacity = totalCapacity.sub(stakerCap);
265         afterResignState.userWithdrawnAfterResign[msg.sender] = true;
266
267         factory.burnLpToken(msg.sender, stakerCap);
268         payable(msg.sender).transfer(stakerCap);
269     } else {
270         require(afterResignState.isWithdrawResignLock, "vPool: Withdraw full
271 capacity is not available yet");
272         factory.burnLpToken(msg.sender, stakerCap);
273         totalCapacity = totalCapacity.sub(stakerCap);
274
275         user.capacity = 0;
276         payable(msg.sender).transfer(stakerCap);
277     }
278
279     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
280
281     emit WithdrawStakeAfterResign(msg.sender, stakerCap);
282 }
```

5.4. Lack of Zero Withdrawal Handling

ID	IDX-004
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	<p>Severity: High</p> <p>Impact: High Loss of user's staked assets can occur if someone calls the <code>resign()</code> function when the <code>totalCapacity</code> is exactly 50,000 VIC due to the revert of the <code>withdrawFromValidator()</code> function call.</p> <p>Likelihood: Medium It is unlikely that a user may call the <code>resign()</code> function when the <code>totalCapacity</code> is exactly 50,000 VIC. However, if the pool is expected to be resigned, stakers are likely to call the <code>unstake()</code> function to withdraw their assets, avoiding the need to wait for resign. This could result in the <code>totalCapacity</code> being close to 50,000 VIC. The attacker can unstake a specific capacity to intentionally set <code>totalCapacity</code> to 50,000.</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by handling the zero withdrawal scenario as suggested.</p>

5.4.1. Description

In `Vpool` contract, the `resign()` function is used to unlock staked VIC and resigning the pool's candidate from `TomoValidator` contract by unvoting from `TomoValidator` contract before resigning to prevent the need to wait for 30 days to withdraw.

Since the `TomoValidator` require to lock the minimum staked VIC amount for 50,000 VIC before resigning, this function calculates the amount to unvote as the total capacity minus 50,000 VIC at line 343, then calls the `unvote()` function on the `TomoValidator` contract to unvote the specified amount at line 346.

Pool.sol

```

336 function resign() public canResign {
337     require(poolStatus == PoolStatus.PROPOSED, "vPool: pool is not proposed");
338
339     updatePool();
340     lastBalance = address(this).balance;
341
342     afterResignState.capBeforeResign = totalCapacity;
343     uint256 _unvoteAmount = totalCapacity.sub(50000 ether);

```

```

344
345     // Unvote before resigning prevents the need to wait for 30 days to
withdraw
346     tomoValidator.unvote(coinbase, _unvoteAmount);
347
348     uint256 withdrawBlockNumber = unvoteDelayedBlocks.add(block.number);
349     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(_unvoteAmount);
350     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
351     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
352     currentWithdrawIndex = currentWithdrawIndex.add(1);
353
354     tomoValidator.resign(coinbase);
355
356     withdrawBlockNumber = resignDelayedBlocks.add(block.number);
357     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(
358         totalCapacity.sub(_unvoteAmount)
359     );
360     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
361     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
362     currentWithdrawIndex = currentWithdrawIndex.add(1);
363
364     poolStatus = PoolStatus.RESIGNED;
365     resignBlock = block.number;
366
367     emit Resign(address(this));
368 }

```

In this case, if the user calls `resign()` function while `totalCapacity` is precisely 50,000 VIC, this function will invoke the `unvote()` function on the `TomoValidator` with an amount of 0. It will then update the `withdrawsState` mapping with a withdrawal cap of 0 in the `TomoValidator` contract at line 218.

Validator.sol

```

212 function unvote(address _candidate, uint256 _cap) public
onlyValidVote(_candidate, _cap) {
213     validatorsState[_candidate].cap =
validatorsState[_candidate].cap.sub(_cap);
214     validatorsState[_candidate].voters[msg.sender] =
validatorsState[_candidate].voters[msg.sender].sub(_cap);
215
216     // refund after delay X blocks
217     uint256 withdrawBlockNumber = voterWithdrawDelay.add(block.number);
218     withdrawsState[msg.sender].caps[withdrawBlockNumber] =
withdrawsState[msg.sender].caps[withdrawBlockNumber].add(_cap);
219     withdrawsState[msg.sender].blockNumbers.push(withdrawBlockNumber);

```

```

220
221     emit Unvote(msg.sender, _candidate, _cap);
222 }

```

The `withdrawFromValidator()` function, which calls `tomoValidator.withdraw()` function, may encounter a revert during the `onlyValidWithdraw` modifier check in `Validator` contract at line 285. This is because the modifier ensures that only withdrawals with caps greater than 0 are allowed.

Pool.sol

```

281 function withdrawFromValidator() public {
282     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&
!afterResignState.isWithdrawUnstakeLock) {
283         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];
284
285         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
286         afterResignState.isWithdrawUnstakeLock = true;
287     }
288
289     if (block.number > resignBlock.add(resignDelayedBlocks) &&
!afterResignState.isWithdrawResignLock) {
290         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];
291
292         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
293         afterResignState.isWithdrawResignLock = true;
294     }
295 }

```

Validator.sol

```

243 function withdraw(uint256 _blockNumber, uint _index) public
onlyValidWithdraw(_blockNumber, _index) {
244     uint256 cap = withdrawsState[msg.sender].caps[_blockNumber];
245     delete withdrawsState[msg.sender].caps[_blockNumber];
246     delete withdrawsState[msg.sender].blockNumbers[_index];
247     msg.sender.transfer(cap);
248     emit Withdraw(msg.sender, _blockNumber, cap);
249 }

```

Validator.sol

```

120 modifier onlyValidWithdraw (uint256 _blockNumber, uint _index) {
121     require(_blockNumber > 0);
122     require(block.number >= _blockNumber);
123     require(withdrawsState[msg.sender].caps[_blockNumber] > 0);
124     require(withdrawsState[msg.sender].blockNumbers[_index] == _blockNumber);

```

```
125     -;  
126 }
```

As a result, users won't be able to call `withdrawAfterResign()` to withdraw their staked assets after the pool has been resigned due to a revert in the `withdrawFromValidator()` function at line 255.

Pool.sol

```
241 function withdrawAfterResign() public nonReentrant {  
242     UserInfo storage user = users[msg.sender];  
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");  
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:  
Withdraw is not available yet");  
245     uint256 stakerCap = user.capacity;  
246  
247     updatePool();  
248     uint256 pending =  
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);  
249  
250     if (pending > 0) {  
251         payable(msg.sender).transfer(pending);  
252         emit WithdrawReward(msg.sender, pending);  
253     }  
254  
255     withdrawFromValidator();  
256  
257     if (  
258         afterResignState.isWithdrawUnstakeLock &&  
!afterResignState.isWithdrawResignLock &&  
!afterResignState.userWithdrawnAfterResign[msg.sender]  
259     ) {  
260         stakerCap = afterResignState.capBeforeResign.sub(50000  
ether).mul(stakerCap).div(afterResignState.capBeforeResign);  
261         user.capacity = user.capacity.sub(stakerCap);  
262         totalCapacity = totalCapacity.sub(stakerCap);  
263         afterResignState.userWithdrawnAfterResign[msg.sender] = true;  
264  
265         factory.burnLpToken(msg.sender, stakerCap);  
266         payable(msg.sender).transfer(stakerCap);  
267     } else {  
268         factory.burnLpToken(msg.sender, stakerCap);  
269         totalCapacity = totalCapacity.sub(stakerCap);  
270  
271         user.capacity = 0;  
272         payable(msg.sender).transfer(stakerCap);  
273     }  
274  
275     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
```

```
276  
277     emit WithdrawStakeAfterResign(msg.sender, stakerCap);  
278 }
```

5.4.2. Remediation

Inspex suggests skipping the withdrawal from the validator when the withdrawal capacity for unvoted blocks of the pool is zero in the `withdrawFromValidator()` function. For example in lines 285 - 287:

Pool.sol

```
281 function withdrawFromValidator() public {  
282     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&  
!afterResignState.isWithdrawUnstakeLock) {  
283         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];  
284  
285         if (withdrawsState[address(this)].caps[_blockNumber] != 0) {  
286             tomoValidator.withdraw(_blockNumber,  
withdrawIndexes[_blockNumber].sub(1));  
287         }  
288         afterResignState.isWithdrawUnstakeLock = true;  
289     }  
290  
291     if (block.number > resignBlock.add(resignDelayedBlocks) &&  
!afterResignState.isWithdrawResignLock) {  
292         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];  
293  
294         tomoValidator.withdraw(_blockNumber,  
withdrawIndexes[_blockNumber].sub(1));  
295         afterResignState.isWithdrawResignLock = true;  
296     }  
297 }
```

5.5. Incorrect Period Validation in the withdrawFromValidator() Function

ID	IDX-005
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Medium</p> <p>Impact: Medium The user can immediately withdraw the entire fund portion without needing to wait for the resignation delay period.</p> <p>Likelihood: Medium The user must execute the <code>withdrawAfterResign()</code> function in the block that <code>block.number == resignBlock.add(unvoteDelayedBlocks)</code> in order to skip the resign delay period</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by using the <code>>=</code> operator to validate the withdrawal period instead.</p>

5.5.1. Description

In the `Vpool` contract, the `withdrawAfterResign()` function there is a period validation that the `block.number >= resignBlock.add(unvoteDelayedBlocks)` at line 244 which check the valid period for withdraw the staked fund. This function also has a call to the `withdrawFromValidator()` function at line 255.

Pool.sol

```

241 function withdrawAfterResign() public nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245     uint256 stakerCap = user.capacity;
246
247     updatePool();
248     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
249
250     if (pending > 0) {
251         payable(msg.sender).transfer(pending);
252         emit WithdrawReward(msg.sender, pending);

```

```

253     }
254
255     withdrawFromValidator();
256
257     if (
258         afterResignState.isWithdrawUnstakeLock &&
!afterResignState.isWithdrawResignLock &&
!afterResignState.userWithdrawnAfterResign[msg.sender]
259     ) {
260         stakerCap = afterResignState.capBeforeResign.sub(50000
ether).mul(stakerCap).div(afterResignState.capBeforeResign);
261         user.capacity = user.capacity.sub(stakerCap);
262         totalCapacity = totalCapacity.sub(stakerCap);
263         afterResignState.userWithdrawnAfterResign[msg.sender] = true;
264
265         factory.burnLpToken(msg.sender, stakerCap);
266         payable(msg.sender).transfer(stakerCap);
267     } else {
268         factory.burnLpToken(msg.sender, stakerCap);
269         totalCapacity = totalCapacity.sub(stakerCap);
270
271         user.capacity = 0;
272         payable(msg.sender).transfer(stakerCap);
273     }
274
275     user.rewardDebt = user.capacity.mul(accRewardPerShare).div(PRECISION);
276
277     emit WithdrawStakeAfterResign(msg.sender, stakerCap);
278 }

```

However, the period validation in the `withdrawFromValidator()` function is using the `>` operator, not the `>=` as shown in lines 282 and 289.

Pool.sol

```

281 function withdrawFromValidator() public {
282     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&
!afterResignState.isWithdrawUnstakeLock) {
283         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];
284
285         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
286         afterResignState.isWithdrawUnstakeLock = true;
287     }
288
289     if (block.number > resignBlock.add(resignDelayedBlocks) &&
!afterResignState.isWithdrawResignLock) {
290         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];

```

```
291
292     tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
293     afterResignState.isWithdrawResignLock = true;
294 }
295 }
```

This allows the user to call the `withdrawAfterResign()` function at the block number `resignBlock.add(unvoteDelayedBlocks)` for skipping the `withdrawFromValidator()` execution, resulting in the `isWithdrawUnstakeLock` flag not being set to true and the user being able to immediately withdraw the entire capacity.

5.5.2. Remediation

Inspex suggests implementing the withdrawal period validation in the `withdrawFromValidator()` function by using `>=` instead.

Pool.sol

```
281 function withdrawFromValidator() public {
282     if (block.number >= resignBlock.add(unvoteDelayedBlocks) &&
!afterResignState.isWithdrawUnstakeLock) {
283         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];
284
285         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
286         afterResignState.isWithdrawUnstakeLock = true;
287     }
288
289     if (block.number >= resignBlock.add(resignDelayedBlocks) &&
!afterResignState.isWithdrawResignLock) {
290         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];
291
292         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
293         afterResignState.isWithdrawResignLock = true;
294     }
295 }
```

5.6. Centralized Authority Control

ID	IDX-006
Target	Vfactory
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	<p>Severity: Medium</p> <p>Impact: High</p> <p>The controlling authorities have the ability to manipulate critical state variables, altering the contract's behavior and potentially increasing their profits. This manipulation can lead to malfunctions and render the system unreliable, which is unfair to platform users.</p> <p>Likelihood: Low</p> <p>The private key of the controlling authorities is unlikely to be compromised. Nevertheless, if it were to be compromised, there are no restrictions preventing unauthorized changes from being made.</p>
Status	<p>Resolved</p> <p>The deFusion team has resolved the issue by implementing timelock mechanism.</p>

5.6.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no mechanism in place to prevent the authorities from altering these variables without notifying the users. In the event that the private key of the controlling authorities is compromised by an attacker, they can manipulate the contract's behavior and profit without any restrictions.

The following `setImplement()` function can be used to set the implementation logic of the pool created by the `Vfactory` contract:

Factory.sol

```
27 function setImplement(address implement) external override onlyOwner {
28     _setImplement(implement);
29
30     emit SetImplement(implement);
31 }
```

5.6.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modification is needed, Inspex suggests implementing a community-run smart contract

governance to control the use of this function.

If removing the function or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time or using the multi-signature contract to reduce the risk of unauthorized or malicious alterations to the smart contract.

5.7. Improper Input Validation

ID	IDX-007
Target	Vpool
Category	General Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	<p>Severity: Low</p> <p>Impact: Low The function can be invoked in contexts not initially designed for its use. This could be used with another vulnerability to escalate the overall impact.</p> <p>Likelihood: Medium There are no restrictions in place to prevent this from being executed. However, exploiting this vulnerability alone lacks incentive, and its impact is more significant when used in conjunction with another exploit.</p>
Status	<p>Resolved The deFusion team has resolved the issue by implementing the input validations as suggested.</p>

5.7.1. Description

Typically, the `unstake()`, `withdrawAfterResign()`, and `withdrawReward()` should only be invoked by users who possess stake tokens on the platform. Nevertheless, these functions are currently accessible to anyone without such restrictions.

Pool.sol

```

171 function unstake(uint256 _amount) external nonReentrant {
172     UserInfo storage user = users[msg.sender];
173     require(user.capacity >= _amount, "vPool: Insufficient amount to
withdraw");
174     require(poolStatus != PoolStatus.RESIGNED, "vPool: Pool is resigned");
175
176     // unstake logic

```

Pool.sol

```

241 function withdrawAfterResign() public nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245

```

```
246 // withdraw after resign logic
```

Pool.sol

```
298 function withdrawReward() public nonReentrant {
299     UserInfo storage user = users[msg.sender];
300
301     updatePool();
302     uint256 pending =
303     user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
304     // withdraw reward logic
```

Moreover, the `withdrawFromValidator()` function should ideally be called only when the pool is in the resigned state. However, it currently lacks any corresponding restrictions.

Pool.sol

```
281 function withdrawFromValidator() public {
282     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&
283     !afterResignState.isWithdrawUnstakeLock) {
284         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];
285         tomoValidator.withdraw(_blockNumber,
286         withdrawIndexes[_blockNumber].sub(1));
287         afterResignState.isWithdrawUnstakeLock = true;
288     }
289     if (block.number > resignBlock.add(resignDelayedBlocks) &&
290     !afterResignState.isWithdrawResignLock) {
291         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];
292         tomoValidator.withdraw(_blockNumber,
293         withdrawIndexes[_blockNumber].sub(1));
294         afterResignState.isWithdrawResignLock = true;
295     }
296 }
```

5.7.2. Remediation

Inspex suggests implementing the `onlyStaker` modifier to the `unstake()`, `withdrawAfterResign()`, and `withdrawReward()` as follows:

Pool.sol

```
171 function unstake(uint256 _amount) external onlyStaker nonReentrant {
172     UserInfo storage user = users[msg.sender];
173     require(user.capacity >= _amount, "vPool: Insufficient amount to
```

```

withdraw");
174     require(poolStatus != PoolStatus.RESIGNED, "vPool: Pool is resigned");
175
176     // unstake logic

```

Pool.sol

```

241 function withdrawAfterResign() public onlyStaker nonReentrant {
242     UserInfo storage user = users[msg.sender];
243     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
244     require(block.number >= resignBlock.add(unvoteDelayedBlocks), "vPool:
Withdraw is not available yet");
245
246     // withdraw after resign logic

```

Pool.sol

```

298 function withdrawReward() public onlyStaker nonReentrant {
299     UserInfo storage user = users[msg.sender];
300
301     updatePool();
302     uint256 pending =
user.capacity.mul(accRewardPerShare).div(PRECISION).sub(user.rewardDebt);
303
304     // withdraw reward logic

```

Inspex suggests implementing the state validation for the `withdrawFromValidator()` function as follows:

Pool.sol

```

281 function withdrawFromValidator() public {
282     require(poolStatus == PoolStatus.RESIGNED, "vPool: Pool is not resigned");
283     if (block.number > resignBlock.add(unvoteDelayedBlocks) &&
!afterResignState.isWithdrawUnstakeLock) {
284         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[0];
285
286         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
287         afterResignState.isWithdrawUnstakeLock = true;
288     }
289
290     if (block.number > resignBlock.add(resignDelayedBlocks) &&
!afterResignState.isWithdrawResignLock) {
291         uint256 _blockNumber = withdrawsState[address(this)].blockNumbers[1];
292
293         tomoValidator.withdraw(_blockNumber,
withdrawIndexes[_blockNumber].sub(1));
294         afterResignState.isWithdrawResignLock = true;
295     }

```

296 }

5.8. Insufficient Check in the propose() Function

ID	IDX-008
Target	Vpool
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	Severity: Very Low Impact: Low In certain situations, the current pool candidate may become unresignable. Likelihood: Low The attacker needs to transfer native tokens directly to the Vpool contract while it is still in the pending state, ensuring that the balance is 50,000 VIC while the totalCapacity remains less than 50,000 VIC then execute the propose() function.
Status	Resolved The deFusion team has resolved the issue by checking the totalCapacity instead, as suggested.

5.8.1. Description

In the **Vpool** contract, the **propose()** function uses the **address(this).balance** to validate whether the contract balance is enough to propose the candidate.

Pool.sol

```
321 function propose() public onlyStaker {
322     require(address(this).balance >= 50000 ether, "vPool: Not enough cap for
propose, minimum 50000 TOMO");
323     require(poolStatus == PoolStatus.PENDING, "vPool: pool is already
proposed");
324
325     tomoValidator.propose{ value: address(this).balance }(coinbase);
326
327     poolStatus = PoolStatus.PROPOSED;
328
329     updatePool();
330     lastBalance = address(this).balance;
331
332     emit Propose(address(this));
333 }
```

However, this allows the attacker to directly transfer the native token to the contract and execute the `propose()` function with the `totalCapacity` being less than 50,000 VIC. This will result in a revert by integer underflow in the `resign()` function when the `totalCapacity.sub(50000 ether)` at line 343.

Pool.sol

```
336 function resign() public canResign {
337     require(poolStatus == PoolStatus.PROPOSED, "vPool: pool is not proposed");
338
339     updatePool();
340     lastBalance = address(this).balance;
341
342     afterResignState.capBeforeResign = totalCapacity;
343     uint256 _unvoteAmount = totalCapacity.sub(50000 ether);
344
345     // Unvote before resigning prevents the need to wait for 30 days to
withdraw
346     tomoValidator.unvote(coinbase, _unvoteAmount);
347
348     uint256 withdrawBlockNumber = unvoteDelayedBlocks.add(block.number);
349     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(_unvoteAmount);
350     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
351     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
352     currentWithdrawIndex = currentWithdrawIndex.add(1);
353
354     tomoValidator.resign(coinbase);
355
356     withdrawBlockNumber = resignDelayedBlocks.add(block.number);
357     withdrawsState[address(this)].caps[withdrawBlockNumber] =
withdrawsState[address(this)].caps[withdrawBlockNumber].add(
358         totalCapacity.sub(_unvoteAmount)
359     );
360     withdrawsState[address(this)].blockNumbers.push(withdrawBlockNumber);
361     withdrawIndexes[withdrawBlockNumber] = currentWithdrawIndex;
362     currentWithdrawIndex = currentWithdrawIndex.add(1);
363
364     poolStatus = PoolStatus.RESIGNED;
365     resignBlock = block.number;
366
367     emit Resign(address(this));
368 }
```

5.8.2. Remediation

Inspex suggests using the `totalCapacity` instead of using the current balance at lines 322 and 325.

Pool.sol

```
321 function propose() public onlyStaker {
322     require(totalCapacity >= 50000 ether, "vPool: Not enough cap for propose,
minimum 50000 TOMO");
323     require(poolStatus == PoolStatus.PENDING, "vPool: pool is already
proposed");
324
325     tomoValidator.propose{ value: totalCapacity }(coinbase);
326
327     poolStatus = PoolStatus.PROPOSED;
328
329     updatePool();
330     lastBalance = address(this).balance;
331
332     emit Propose(address(this));
333 }
```

5.9. Outdated Compiler Version

ID	IDX-009
Target	Vfactory Vpool Ownable Payable ProxyFactory SafeMath
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	Severity: Very Low Impact: Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. Likelihood: Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	Resolved The deFusion team has resolved the issue by using the Solidity compiler version 0.8.19.

5.9.1. Description

The Solidity compiler versions specified in the smart contracts were outdated (<https://soliditylang.org/blog/2023/11/08/solidity-0.8.23-release-announcement>). As the compilers are regularly updated with bug fixes and new features, the latest stable compiler version should be used to compile the smart contracts for best practice.

Pool.sol

1	// SPDX-License-Identifier: Apache-2.0
2	pragma solidity 0.8.16;

The table below represents the contracts that apply the outdated Solidity compiler version.

File	Version
Factory.sol (L:2)	0.8.16
Pool.sol (L:2)	0.8.16

Ownable.sol (L:2)	0.8.16
Payable.sol (L:2)	0.8.16
ProxyFactory.sol (L:2)	0.8.16
SafeMath.sol (L:2)	0.8.16

5.9.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version (<https://github.com/ethereum/solidity/releases>). At the time of audit, the latest stable version of the Solidity compiler in major 0.8 is 0.8.23.

For chains that may not be compatible with Solidity compiler version 0.8.23, Inspex suggests using Solidity compiler version 0.8.19 instead, as Solidity compiler version 0.8.20 or later introduces the PUSH0 (0x5f) opcode, which some chains have not yet included.

Pool.sol

```
1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity 0.8.19;
```

5.10. Incorrect Event Logging

ID	IDX-010
Target	Vfactory Vpool
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	Severity: Info Impact: None Likelihood: None
Status	Resolved The deFusion team has resolved the issue by correcting the event logging.

5.10.1. Description

Incorrect event logging has the potential to mislead users and developers, as it may create confusion or provide inaccurate information about the state or actions within a smart contract. For example, the emitted `PoolCreated` event at line 51 is inconsistent with the event declared in the `IFactory` interface, which specifies `PoolCreated(address _poolAddress, string _poolName, address _coinbase)`.

Vfactory.sol

```
40 function createPool(  
41     bytes32 salt,  
42     string memory _poolName,  
43     address _coinbase  
44 ) public returns (address pool) {  
45     pool = _cloneProxy(salt);  
46     IPool(pool).init(_poolName, _coinbase);  
47     Ownable(pool).transferOwnership(address(this));  
48     address _poolAddress = address(pool);  
49     allPools.push(_poolAddress);  
50     isPool[_poolAddress] = true;  
51     emit PoolCreated(_poolAddress, salt, _coinbase);  
52 }
```

The following table contains all events that emit an incorrect event.

File	Contract	Event
Factory.sol (L: 51)	Vfactory	PoolCreated()
Pool.sol (L: 332)	Vpool	Propose()
Pool.sol (L: 367)	Vpool	Resign()

5.10.2. Remediation

Inspex suggests correcting the event emitting to ensure that the emitted event contains accurate information aligning with the business requirements.

5.11. Inexplicit Solidity Compiler Version

ID	IDX-011
Target	Context
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	Severity: Info Impact: None Likelihood: None
Status	Resolved The deFusion team has resolved the issue by fixing the Solidity compiler version to 0.8.19.

5.11.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

Context.sol

```
1 // SPDX-License-Identifier: MIT
2 // OpenZeppelin Contracts v4.4.1 (utils/Context.sol)
3
4 pragma solidity ^0.8.0;
```

5.11.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.23. (<https://github.com/ethereum/solidity/releases>)

For chains that may not be compatible with Solidity compiler version 0.8.23, Inspex suggests using Solidity compiler version 0.8.19 instead, as Solidity compiler version 0.8.20 or later introduces the PUSH0 (0x5f) opcode, which some chains have not yet included.

Context.sol

```
1 // SPDX-License-Identifier: MIT
2 // OpenZeppelin Contracts v4.4.1 (utils/Context.sol)
3
4 pragma solidity 0.8.19;
```

5.12. Improper Function Visibility

ID	IDX-012
Target	Vfactory Vpool
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding
Risk	Severity: Info Impact: None Likelihood: None
Status	Resolved The deFusion team has resolved the issue by changing functions' visibility to external.

5.12.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

The following source code shows that the `createPool()` function of the `Vfactory` contract is set to public and it is never called from any internal function.

Factory.sol

```
40 function createPool(  
41     bytes32 salt,  
42     string memory _poolName,  
43     address _coinbase  
44 ) public returns (address pool) {  
45     pool = _cloneProxy(salt);  
46     IPool(pool).init(_poolName, _coinbase);  
47     Ownable(pool).transferOwnership(address(this));  
48     address _poolAddress = address(pool);  
49     allPools.push(_poolAddress);  
50     isPool[_poolAddress] = true;  
51     emit PoolCreated(_poolAddress, salt, _coinbase);  
52 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

File	Contract	Function
Factory.sol (L: 40)	Vfactory	createPool()
Pool.sol (L: 95)	Vpool	getWithdrawBlockNumbers()
Pool.sol (L: 101)	Vpool	currentStakerReward()
Pool.sol (L: 116)	Vpool	isVoted()
Pool.sol (L: 219)	Vpool	withdrawUnstakedAmount()
Pool.sol (L: 241)	Vpool	withdrawAfterResign()
Pool.sol (L: 298)	Vpool	withdrawReward()
Pool.sol (L: 316)	Vpool	getCoinbaseAddress()
Pool.sol (L: 321)	Vpool	propose()
Pool.sol (L: 336)	Vpool	resign()

5.12.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

Factory.sol

```

40 function createPool(
41     bytes32 salt,
42     string memory _poolName,
43     address _coinbase
44 ) external returns (address pool) {
45     pool = _cloneProxy(salt);
46     IPool(pool).init(_poolName, _coinbase);
47     Ownable(pool).transferOwnership(address(this));
48     address _poolAddress = address(pool);
49     allPools.push(_poolAddress);
50     isPool[_poolAddress] = true;
51     emit PoolCreated(_poolAddress, salt, _coinbase);
52 }
```

6. Appendix

6.1. About Inspex



CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement



inspex
CYBERSECURITY PROFESSIONAL SERVICE