Security Audit Report for Neptune Mutual Cover

Date: June 21, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.06.21 15:56:57 +08'00'

Contents

Report Manifest

Item	Description
Client	Neptune
Target	Neptune Mutual Cover

Version History

Version	Date	Description
1.0	June 21, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

Neptune Mutual is an on-chain insurance project. Basically, it allows other DApp projects to create "covers", and users can buy these "covers" by paying fees and receive "cxTokens". Note that an IPFS URL needs to be specified to create a cover, and this url is used to indicate the content of the cover (e.g., the list of events which will be covered). Therefore, when a security incident occurs (e.g., a project protected by the cover is hacked with some losses), it will be reported on-chain and determined by a consensus procedure, i.e., users can lock their NPM tokens to vote for or against the validity of this report. If the validity is accepted, each "cxToken" can be claimed to a single unit of stablecoin [1]. Furthermore, to support the claim procedure, liquidity providers (LPs) can provide liquidity to the liquidity vaults created for the covers. The fees paid by cover purchasers are also transferred to these vaults. If no incident happens, the LPs earn the fees; otherwise, liquidity from LPs will be distributed to cover purchasers as the compensation. Namely, it means that unlike most DeFi protocols, the LPs are subject to losses. Finally, all the liquidity in the vaults is not locked in the contract. Instead, a PCV-like strategy is employed so that a portion of the available liquidity in the pool is deposited into AAVE and Compound to earn interests, and the interests are distributed proportionally to the LPs. Apart from that, the vaults also support flash loans to earn fees for LPs.

As the Neptune project is a big project with a large number of contracts and complicated inter-dependencies, the auditors assume that:

- The contract addresses and roles are set correctly.
- Some duplication of checks (for example, some access control requirements are checked twice) are ignored if they do not affect the semantics of the code.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [2] during the audit are shown in the following. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report. Note that, we did **NOT** audit all the modules in the repository. Only the newest version of the contracts are within the audit scope.

Project		Commit SHA
Neptune Mutual Cover	Version 1	d3abb479c0c9e1c972430d4113408ddfc20be5b5
	Version 2	de4e313dd6e8076454b6e5998bb739d897439253

[1] E.g., DAI in Ethereum, which means 1 "cxToken" can be claimed into 1 DAI.

[2] https://github.com/neptune-mutual-blue/protocol

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

- **Fixed** The issue has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **thirteen** potential issues. Besides, we also have **two** recommendations and **one** note.

- High Risk: 2
- Medium Risk: 5
- Low Risk: 6
- Recommendation: 2
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Shadowed variables	Software Security	Fixed
2	Medium	Unchecked input parameters	Software Security	Fixed
3	Low	Conflict access control checks in the `updateCoverUsersWhitelist` function	Software Security	Fixed
4	High	Incentivization design problems	DeFi Security	Fixed
5	High	Potential oracle manipulation	DeFi Security	Fixed
6	Low	Potential uninitialized price information	DeFi Security	Acknowledged
7	Low	Incorrect check of the return value of the `transfer` function	DeFi Security	Fixed
8	Low	Incorrect logic in the `removeLiquidity` function	DeFi Security	Fixed
9	Medium	Incorrect handling of LP tokens in the `removeLiquidity` function	DeFi Security	Fixed
10	Medium	Potential less reward distributed to the first reporter	DeFi Security	Acknowledged
11	Medium	Potential fee rate manipulation	DeFi Security	Fixed
12	Medium	Incorrect logic of calculating the deposit amount for strategies	DeFi Security	Fixed
13	Low	No fixed voting reward claim period for false reporting	DeFi Security	Acknowledged
14	-	Remove the redundant calculation	Recommendation	Fixed
15	-	Remove the debug logs	Recommendation	Acknowledged
16	-	Potential centrality problems	Note	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Shadowed variables

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The design of the Neptune contract system has a root `Store` contract that other contracts store all the related data to this contract, scoped by different prefixes and hashed keys. Both the `deploy`

function of `cxTokenFactory` and `VaultFactory` contracts have a parameter `s` (which is a reference to the core `Store` contract) that shadows the state variable `s` (which is also a reference to the `Store` contract, but is inherited from the parent contract `Recoverable`). As these two variables have exactly the same references, one of them should be removed.

```
36    function deploy(
37        IStore s,
38        bytes32 key,
39        uint256 expiryDate
40      ) external override nonReentrant returns (address deployed) {
41        // @suppress-acl Can only be called by the latest policy contract
42        s.mustNotBePaused();
43        s.mustBeValidCoverKey(key);
44        s.senderMustBePolicyContract();
45
46        require(expiryDate > 0, "Please specify expiry date");
47
48        (bytes memory bytecode, bytes32 salt) = cxTokenFactoryLibV1.getByteCode(s, key, expiryDate)
              ;
49
50        require(s.getAddress(salt) == address(0), "Already deployed");
51
52        // solhint-disable-next-line
53        assembly {
54          deployed := create2(
55            callvalue(), // wei sent with current call
56            // Actual code starts after skipping the first 32 bytes
57            add(bytecode, 0x20),
58            mload(bytecode), // Load the size of code contained in the first 32 bytes
59            salt // Salt from function arguments
60          )
61
62          if iszero(extcodesize(deployed)) {
63            // @suppress-revert This is correct usage
64            revert(0, 0)
65          }
66        }
67
68        s.setAddress(salt, deployed);
69        s.setBoolByKeys(ProtoUtilV1.NS_COVER_CXTOKEN, deployed, true);
70        s.setAddressArrayByKeys(ProtoUtilV1.NS_COVER_CXTOKEN, key, deployed);
71
72        emit CxTokenDeployed(key, deployed, expiryDate);
73      }
```

Listing 2.1: cxTokenFactory.sol

```
34  function deploy(IStore s, bytes32 key) external override nonReentrant returns (address addr) {
35    s.mustNotBePaused();
36    s.mustHaveNormalCoverStatus(key);
37    s.senderMustBeCoverContract();
38
39    (bytes memory bytecode, bytes32 salt) = VaultFactoryLibV1.getByteCode(s, key, s.getStablecoin
```

```
        ());
40
41    // solhint-disable-next-line
42    assembly {
43      addr := create2(
44        callvalue(), // wei sent with current call
45        // Actual code starts after skipping the first 32 bytes
46        add(bytecode, 0x20),
47        mload(bytecode), // Load the size of code contained in the first 32 bytes
48        salt // Salt from function arguments
49      )
50
51      if iszero(extcodesize(addr)) {
52        // @suppress-revert This is correct usage
53        revert(0, 0)
54      }
55    }
56
57    emit VaultDeployed(key, addr);
58  }
```

Listing 2.2: VaultFactory.sol

```
 9    abstract contract Recoverable is ReentrancyGuard, IRecoverable {
10        using ValidationLibV1 for IStore;
11        IStore public override s;
```

Listing 2.3: Recoverable.sol

Impact N/A

Suggestion Remove one of the references of the `Store` contract.

2.1.2 Unchecked input parameters

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_addCover` function of the `CoverLibV1` library, some critical parameters are used to create covers. Note that these parameters are important because they would affect the behavior of the created covers. For example, it is allowed to set the claim period to be zero. However, although in other situations these parameters are verified, they are not checked in the `_addCover` function.

```
124    function _addCover(
125        IStore s,
126        bytes32 key,
127        bytes32 info,
128        address reassuranceToken,
129        bool requiresWhitelist,
130        uint256[] memory values,
131        uint256 fee
132      ) private {
```

```
133         s.setBoolByKeys(ProtoUtilV1.NS_COVER, key, true);
134
135         s.setStatusInternal(key, 0, CoverUtilV1.CoverStatus.Stopped);
136
137         s.setAddressByKeys(ProtoUtilV1.NS_COVER_OWNER, key, msg.sender);
138         s.setBytes32ByKeys(ProtoUtilV1.NS_COVER_INFO, key, info);
139         s.setAddressByKeys(ProtoUtilV1.NS_COVER_REASSURANCE_TOKEN, key, reassuranceToken);
140         s.setUintByKeys(ProtoUtilV1.NS_COVER_REASSURANCE_WEIGHT, key, ProtoUtilV1.MULTIPLIER); //
                100% weight because it's a stablecoin
141         s.setBoolByKeys(ProtoUtilV1.NS_COVER_REQUIRES_WHITELIST, key, requiresWhitelist);
142
143         // Set the fee charged during cover creation
144         s.setUintByKeys(ProtoUtilV1.NS_COVER_FEE_EARNING, key, fee);
145
146         s.setUintByKeys(ProtoUtilV1.NS_GOVERNANCE_REPORTING_MIN_FIRST_STAKE, key, values[2]);
147         s.setUintByKeys(ProtoUtilV1.NS_GOVERNANCE_REPORTING_PERIOD, key, values[3]);
148         s.setUintByKeys(ProtoUtilV1.NS_RESOLUTION_COOL_DOWN_PERIOD, key, values[4]);
149         s.setUintByKeys(ProtoUtilV1.NS_CLAIM_PERIOD, key, values[5]);
150         s.setUintByKeys(ProtoUtilV1.NS_COVER_POLICY_RATE_FLOOR, key, values[6]);
151         s.setUintByKeys(ProtoUtilV1.NS_COVER_POLICY_RATE_CEILING, key, values[7]);
152     }
```

Listing 2.4: CoverLibV1.sol

Impact These important parameters might be abused.

Suggestion Add proper checks to verify these parameters.

2.1.3 Conflict access control checks in the `updateCoverUsersWhitelist` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The cover creators are allowed to set a whitelist so that only the whitelisted users can purchase covers. The creators are allowed to update this whitelist through the `updateCoverUsersWhitelist` function. However, there is a conflict check in the `updateCoverUsersWhitelist` function that requires the caller to be both the cover manager and the cover owner/admin.

```
148     function updateCoverUsersWhitelist(
149         bytes32 key,
150         address[] memory accounts,
151         bool[] memory statuses
152     ) external override nonReentrant {
153         s.mustNotBePaused();
154         AccessControlLibV1.mustBeCoverManager(s);
155         s.senderMustBeCoverOwnerOrAdmin(key);
156
157         s.updateCoverUsersWhitelistInternal(key, accounts, statuses);
158     }
```

Listing 2.5: Cover.sol

Impact This function may be inaccessible because of the conflict checks.

Suggestion Remove either of them.

2.2 DeFi Security

2.2.1 Incentivization design problems

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Neptune is an insurance-like project so that anyone is able to buy covers for the project they are holding. There exists a reporting-and-voting mechanism when a security incident occurs (e.g., the covered project is attacked). Specifically, the reporters are responsible for reporting the real world incidents, while the voters are responsible for determining the validity of the reported incidents. Reporters and voters are rewarded for their contributions accordingly.

However, due to the difficulty of verifying the on-chain identities, there are at least two cases where the incentivization design can be abused:

1. A reporter finds that no one has spotted and reported the incident, then the optimal action for him/her is to buy `cxTokens` (maybe through another account) first to earn more.
2. If the attacker's target is covered in Neptune. Then his optimal action would be buying `cxTokens` first, then completing the attack, and finally reporting this incident.

Impact In these cases, the cover has to pay for a "bad" purchaser that has already foreseen the result, which violates the cover design and causes losses to LPs.

Suggestion N/A

Feedback from the Project We have added a new "exclusion" to all covers stating that policies will only be effective after 24 hours of purchase. Or in other words, policies purchased within the last 24 hours (in this case, before reporting starts or the real incident happened) are not valid and therefore excluded. Since anyone can purchase cover at any time, we group "effective date" by taking the UTC EOD (end of the day) timestamp after 24 hours of the policy purchase date.

Since a reporting period of 7-days is started once a report is submitted, this gives us enough time to manually (or in an automated way) to blacklist any malicious users who purchase the protection. Here, we manually set addresses and amounts that are ineligible to receive the payout. Because we are taking "ethereum address" into consideration when implementing blacklist and delayed coverage, we have restricted the transferability of cxTokens.

2.2.2 Potential oracle manipulation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The prices used in the project (e.g., the NPM price) are derived from the Uniswap V2 pair reserves directly, which is subject to price manipulation attacks. Though the reserve information is updated

in an specified interval (e.g., the current price of the Uniswap V2 pair would not be effective after the interval specified by the `NS_LIQUIDITY_STATE_UPDATE_INTERVAL` parameter) to prevent flashloan attacks, it still suffers from the following problems:

- The token prices in the project are delayed, i.e., the current prices used are updated in the previous interval. It would cause the normal price fluctuations cannot be timely reflected into the project.
- This design cannot prevent price manipulation attacks. Specifically, a price manipulation attack can be achieved by the following actions:
  1. The attacker manipulates the reserves in the (NPM, stablecoin) pair.
  2. The normal transaction that triggers the `updateStateAndLiquidity` is executed, updating the manipulated reserves (and hence the manipulated price) into the system.
  3. The attacker exploits the manipulated price (e.g., in the Bonding).

```
107   function getPriceInternal(
108       IStore s,
109       address token,
110       address stablecoin,
111       uint256 multiplier
112   ) public view returns (uint256) {
113       IUniswapV2PairLike pair = _getPair(s, token, stablecoin);
114       IUniswapV2RouterLike router = IUniswapV2RouterLike(s.getUniswapV2Router());
115
116       uint256[] memory values = getLastKnownPairInfoInternal(s, pair);
117       uint256 reserve0 = values[0];
118       uint256 reserve1 = values[1];
119
120       if (pair.token0() == stablecoin) {
121         return router.getAmountIn(multiplier, reserve0, reserve1);
122       }
123
124       return router.getAmountIn(multiplier, reserve1, reserve0);
125   }
```

Listing 2.6: PriceLibV1.sol

Impact The price oracle is subject to price manipulation attacks and may suffer from delayed prices under some extreme circumstances, which may cause the loss of the protocol. For example, the NPM token price is used in both `BondPool` and `cxToken` fee calculation (for provision liquidity calculation).

Suggestion To further solve the price oracle problem, a TWAP price on top of the Uniswap V2 pair is recommended.

2.2.3 Potential uninitialized price information

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description The `BondPool` contract is used to provide the holders of the LP token of a specified Uniswap-like pair to buy the NPM tokens at a discounted price. Under normal assumptions, the LP token is considered to be the pair of the stablecoin and the NPM token. Then the price of this pair is regularly updated and

stored. However, if this LP token refers to other pairs, the pair information would not be properly updated and the transaction would revert.

Impact The pair information may not be updated on time if the LP token refers to other pairs.

Suggestion N/A

Feedback from the Project The Bond Pool feature will ALWAYS have one LP pair (NPM/USDC or NPM/DAI) at one given time, which is less likely to change. The LP pair can change only if we decide that we want to migrate liquidity to a different stablecoin, say from DAI to USDC or vice versa.

2.2.4 Incorrect check of the return value of the `transfer` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Some contracts in the project inherit the `Recoverable` contract so that the project owners can withdraw the full amount of any tokens from the contract. This functionality is implemented so that any funds which are accidentally sent to the contracts can be recovered. Internally, the `recoverTokenInternal` function in the `BaseLibV1` library handles the transfer. In the `recoverTokenInternal` function, the plain ERC-20 transfer function is used and the return value is force checked, which could not handle some special tokens (e.g., USDT) that do not strictly comply with the ERC-20 standard.

```
30    function recoverTokenInternal(address token, address sendTo) external {
31        // @suppress-address-trust-issue, @suppress-malicious-erc20 Although the token can't be
              trusted, the recovery agent has to check the token code manually.
32        IERC20 erc20 = IERC20(token);
33
34        uint256 balance = erc20.balanceOf(address(this));
35
36        if (balance > 0) {
37          require(erc20.transfer(sendTo, balance), "Transfer failed");
38        }
39      }
40    }
```

Listing 2.7: BaseLibV1.sol

Impact The token with incorrect implementation (i.e., it does not return any value in the `transfer` function) cannot be recovered from the protocol.

Suggestion Replace ERC20 transfer with the `SafeTransfer` library of OpenZeppelin.

2.2.5 Incorrect logic in the `removeLiquidity` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `removeLiquidity` function of the `VaultLiquidity` contract, the caller can specify the `exit` parameter. This parameter is used in the `VaultLibV1.preRemoveLiquidityInternal` function to call

the `_unStakeNpm` function, indicating that the user wants to unstake the NPM tokens provided in the adding liquidity procedure. Unfortunately, this implementation has the following two flaws:

1. The stake mechanism might be broken. Note that users must stake some NPM tokens in the `addLiquidity` function. However, it allows a user that has staked NPM in the adding liquidity procedure to withdraw the staked NPM by removing a (small) amount of liquidity and setting the `exit` parameter to true.
2. The behavior of invoking the `removeLiquidity` function with the `exit` parameter set to true seems to be problematic. It allows a user to withdraw part of the staked NPMs. However, such a behavior is far away from the semantic of `exit`.

```
200   function _unStakeNpm(
201       IStore s,
202       address account,
203       bytes32 coverKey,
204       uint256 amount,
205       bool exit
206     ) private {
207       uint256 remainingStake = _getMyNpmStake(s, coverKey, account);
208       uint256 minStakeToMaintain = exit ? 0 : s.getMinStakeToAddLiquidity();
209
210       require(remainingStake - amount >= minStakeToMaintain, "Can't go below min stake");
211
212       s.subtractUintByKey(CoverUtilV1.getCoverLiquidityStakeKey(coverKey), amount); // Total
              stake
213       s.subtractUintByKey(CoverUtilV1.getCoverLiquidityStakeIndividualKey(coverKey, account),
              amount); // Your stake
214     }
```

Listing 2.8: VaultLibV1.sol

Impact The `exit` parameter does not work as expected, and the requirement of staking NPM when adding liquidity can be bypassed.

Suggestion Fix the incorrect logic.

2.2.6 Incorrect handling of LP tokens in the `removeLiquidity` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `removeLiquidity` function of the `VaultLiquidity` contract, after transferring the LP tokens from the LP to the vault, the LP tokens are not burned but kept in the vault. As a result, the `totalSupply` of the LP tokens could not be properly calculated.

```
 95   function removeLiquidity(
 96       bytes32 coverKey,
 97       uint256 podsToRedeem,
 98       uint256 npmStakeToRemove,
 99       bool exit
100     ) external override nonReentrant {
```

```
101       // @suppress-acl Marking this as publicly accessible
102       require(coverKey == key, "Forbidden");
103       require(podsToRedeem > 0, "Please specify amount");
104
105       /*****************************************************************************************
106         PRE
107        *****************************************************************************************
              */
108       (address stablecoin, uint256 stablecoinToRelease) = delgate().preRemoveLiquidity(msg.sender
              , coverKey, podsToRedeem, npmStakeToRemove, exit);
109
110       /*****************************************************************************************
111         BODY
112        *****************************************************************************************
              */
113       IERC20(address(this)).ensureTransferFrom(msg.sender, address(this), podsToRedeem);
114       IERC20(stablecoin).ensureTransfer(msg.sender, stablecoinToRelease);
115
116       // Unstake NPM tokens
117       if (npmStakeToRemove > 0) {
118         IERC20(s.getNpmTokenAddress()).ensureTransfer(msg.sender, npmStakeToRemove);
119       }
120
121       /*****************************************************************************************
122         POST
123        *****************************************************************************************
              */
124       delgate().postRemoveLiquidity(msg.sender, coverKey, podsToRedeem, npmStakeToRemove, exit);
125
126       emit PodsRedeemed(msg.sender, podsToRedeem, stablecoinToRelease);
127
128       if (exit) {
129         emit Exited(coverKey, msg.sender);
130       }
131
132       if (npmStakeToRemove > 0) {
133         emit NPMUnstaken(msg.sender, npmStakeToRemove);
134       }
135     }
```

Listing 2.9: VaultLiquidity.sol

Impact The incorrect `totalSupply` of the LP tokens may cause the loss of the LPs for withdrawal.

Suggestion Burn the LP tokens transferred to the Vault in the `removeLiquidity` function.

2.2.7 Potential less reward distributed to the first reporter

Severity Medium

Status Acknowledged

Introduced by `Version 1`

Description To provide incentives to the reporter, the first reporter (address) that reports the incident has a special reward. Specifically, after an `IncidentHappened` report, voters in the winning camp can claim the voting rewards by invoking the `Unstakable.unstakeWithClaim` function. Every claim would distribute rewards to the voter, the first reporter and the burner address (i.e. some rewards in NPM are burnt), respectively. However, this mechanism has a defect that the first report may receive less than expected if there are voters not claiming their rewards.

```
60  function unstakeWithClaim(bytes32 key, uint256 incidentDate) external override nonReentrant {
61    require(incidentDate > 0, "Please specify incident date");
62
63    // @suppress-acl Marking this as publicly accessible
64    // @suppress-pausable Already checked inside `validateUnstakeWithClaim`
65    s.validateUnstakeWithClaim(key, incidentDate);
66
67    address finalReporter = s.getReporterInternal(key, incidentDate);
68    address burner = s.getBurnAddress();
69
70    (, , uint256 myStakeInWinningCamp, uint256 toBurn, uint256 toReporter, uint256 myReward, ) = s
          .getUnstakeInfoForInternal(msg.sender, key, incidentDate);
71
72    // Set the unstake details
73    s.updateUnstakeDetailsInternal(msg.sender, key, incidentDate, myStakeInWinningCamp, myReward,
          toBurn, toReporter);
74
75    uint256 myStakeWithReward = myReward + myStakeInWinningCamp;
76
77    s.npmToken().ensureTransfer(msg.sender, myStakeWithReward);
78
79    if (toReporter > 0) {
80      s.npmToken().ensureTransfer(finalReporter, toReporter);
81    }
82
83    if (toBurn > 0) {
84      s.npmToken().ensureTransfer(burner, toBurn);
85    }
86
87    s.updateStateAndLiquidity(key);
88
89    emit Unstaken(msg.sender, myStakeInWinningCamp, myReward);
90    emit ReporterRewardDistributed(msg.sender, finalReporter, myReward, toReporter);
91    emit GovernanceBurned(msg.sender, burner, myReward, toBurn);
92  }
```

Listing 2.10: Unstakable.sol

Impact The first reporter may not get all the rewards.

Suggestion N/A

Feedback from the Project This behavior is by design and we will update the documentation to explain this in more detail. The first/honest/resolved reporter receives a cut of reward received by each individual unstaker.

Reference: https://docs.neptunemutual.com/covers/cover-reporting

2.2.8 Potential fee rate manipulation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The factors that are used to calculate the fee rates for buying `cxToken` are subject to manipulation. Specifically, the fee rate is calculated based on the following formula:

$$\texttt{totalAvailableLiquidity} = \texttt{stablecoinOwnedByVault} + \texttt{supportPool} \tag{2.1}$$

$$\texttt{utilizationRatio} = (\texttt{commitment} + \texttt{amountToCover})/\texttt{totalAvailableLiquidity} \tag{2.2}$$

Note that the final fee rate is proportional to `utilizationRatio`. As such, the factors can be manipulated in the following way:

- The NPM price is subject to price manipulation, and finally affects the fee rate. Please see the issue in Section 2.2.2 for potential price manipulation. The manipulation of the NPM price would lead to the manipulation of the supportPool, and hence the fee rate.
- The `stablecoinOwnedByVault` can be manipulated by flashloans. A `cxToken` buyer A can borrow external flashloans and provide liquidity in the withdrawal period. Because within the withdrawal period, adding and removing liquidity can be done in one transaction. Specifically, the strategy of A can be illustrated in the following three steps:
  - Borrowing the flashloan from other platforms and adding liquidity.
  - Buying `cxToken`s with a discount, because the `stablecoinOwnedByVault` is larger with the liquidity added in the first step, which lowers the fee rate.
  - Removing liquidity and paying back the flashloan. Note that the `cxToken` fee generated from the second step is distributed to liquidity providers immediately and proportionally. Due to the liquidity provision in the first step, the fee is also distributed to A. It further lowers the cost for A to buy the covers.

```
22  function calculatePolicyFeeInternal(
23    IStore s,
24    bytes32 key,
25    uint256 coverDuration,
26    uint256 amountToCover
27  )
28    public
29    view
30    returns (
31      uint256 fee,
32      uint256 utilizationRatio,
33      uint256 totalAvailableLiquidity,
34      uint256 floor,
35      uint256 ceiling,
36      uint256 rate
37    )
38  {
39    (floor, ceiling) = getPolicyRatesInternal(s, key);
```

```
40    (uint256 stablecoinOwnedByVault, uint256 commitment, uint256 supportPool) =
          _getCoverPoolAmounts(s, key);
41
42    require(amountToCover > 0, "Please enter an amount");
43    require(coverDuration > 0 && coverDuration <= 3, "Invalid duration");
44    require(floor > 0 && ceiling > floor, "Policy rate config error");
45
46    require(stablecoinOwnedByVault - commitment > amountToCover, "Insufficient fund");
47
48    totalAvailableLiquidity = stablecoinOwnedByVault + supportPool;
49    utilizationRatio = (ProtoUtilV1.MULTIPLIER * (commitment + amountToCover)) /
          totalAvailableLiquidity;
50
51    console.log("[cp] s: %s. p: %s. u: %s", stablecoinOwnedByVault, supportPool, utilizationRatio)
          ;
52    console.log("[cp]: %s, a: %s. t: %s", commitment, amountToCover, totalAvailableLiquidity);
53
54    rate = utilizationRatio > floor ? utilizationRatio : floor;
55
56    rate = rate + (coverDuration * 100);
57
58    if (rate > ceiling) {
59      rate = ceiling;
60    }
61
62    fee = (amountToCover * rate * coverDuration) / (12 * ProtoUtilV1.MULTIPLIER);
63  }
```

Listing 2.11: PolicyHelperV1.sol

Impact The attacker may buy `cxToken`s with an unfair discount.

Suggestion N/A

Feedback from the Project NPM token pricing dependence will be removed along with NPM provision feature. We will update the add and remove liquidity function to have at least one block gap between the two.

2.2.9 Incorrect logic of calculating the deposit amount for strategies

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The stablecoin liquidity deposited in the vaults is further provided into lending projects like AAVE and Compound to earn interest. Specifically, the `RoutineInvokerLibV1` library is used to provide the logic for depositing into different strategies. In the `_canDeposit` function, the current deposit amount and the maximum amount that can be deposited are calculated and compared. The calculation and comparison are used to ensure that sufficient liquidity is preserved before depositing to the lending projects.

However, there is a logic problem in the calculation, as the `_canDeposit` function will be called regardless of the status of the current strategy. Specifically, this function can be invoked out of the withdrawal period. By doing so, the liquidity will not be withdrawn from the strategies, which means the `maximumAllowed`

amount that is calculated based on the current balance of the vault can be small. As a result, some liquidity in the vaults may not be deposited to earn interest.

```
169  function _canDeposit(
170    IStore s,
171    ILendingStrategy strategy,
172    uint256 totalStrategies,
173    bytes32 key
174  ) private view returns (uint256) {
175    address vault = s.getVaultAddress(key);
176    IERC20 stablecoin = IERC20(s.getStablecoin());
177
178    uint256 maximumAllowed = (stablecoin.balanceOf(vault) * s.getMaxLendingRatioInternal()) /
           ProtoUtilV1.MULTIPLIER;
179    uint256 allocation = maximumAllowed / totalStrategies;
180    uint256 weight = strategy.getWeight();
181    uint256 canDeposit = (allocation * weight) / ProtoUtilV1.MULTIPLIER;
182    uint256 alreadyDeposited = s.getAmountInStrategy(key, strategy.getName(), address(stablecoin))
           ;
183
184    if (alreadyDeposited >= canDeposit) {
185      return 0;
186    }
187
188    return canDeposit - alreadyDeposited;
189  }
```

Listing 2.12: RoutineInvokerLibV1.sol

Impact Some liquidity in the vaults might not be deposited to earn interest due to the incorrect calculation in the `canDeposit` function.

Suggestion Fix the incorrect logic.

2.2.10 No fixed voting reward claim period for false reporting

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description If an incident report is voted as false, the state of the cover will remain `FalseReporting` until the finalization is requested by the governance. In contrast, in the case of `IncidentHappened`, a fixed claim period is set to let both the cxToken holders and the voters in the winning camp to claim their compensation and rewards.

Impact Without a fixed claim period, if the state of the cover is finalized by the governance agent, the winning camp of the refuting side cannot withdraw their rewards.

Suggestion Setting another fixed period for the winning camp to claim their voting rewards for the `FalseReporting` incident.

Feedback from the Project When an incident is incorrectly reported by malicious actors who do not care about their reporting fee being forfeited but just want to create nuisance for the cover creators and

our protocol, we want to be able to quickly resolve the incident with as little delay as possible and reset the cover back to the normal status.

In this situation, the amount of time to access the 'unstakeWithClaim' function will be made available on a case to case basis. In other words, based on the the-then situation and request from cover creators, we would manually announce that we would resolve the cover within 24 hours or 48 hours and so that the witnesses should unstake (with claim) as soon as they can or before this time.

We need this flexibility to defend against targeted attacks to specific cover pools. Furthermore, we may customize and shorten the reporting period of individual covers and/or increase the minimum amount of first reporting stake required so that financial risk arising from this attack is bigger and enough to deter unwanted reportings.

2.3 Additional Recommendation

2.3.1 Remove the redundant calculation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `getReportingUnstakenAmountInternal` function of the `GovernanceUtilV1` contract, the key for retrieving the storage value is calculated twice.

```
184    function getReportingUnstakenAmountInternal(
185        IStore s,
186        address account,
187        bytes32 key,
188        uint256 incidentDate
189      ) public view returns (uint256) {
190        bytes32 k = keccak256(abi.encodePacked(ProtoUtilV1.NS_GOVERNANCE_UNSTAKE_TS, key,
               incidentDate, account));
191        k = keccak256(abi.encodePacked(ProtoUtilV1.NS_GOVERNANCE_UNSTAKEN, key, incidentDate,
               account));
192        return s.getUintByKey(k);
193      }
```

Listing 2.13: GovernanceUtilV1.sol

Impact N/A

Suggestion Remove the redundant calculation to save gas.

2.3.2 Remove the debug logs

Status Acknowledged

Introduced by `Version 1`

Description There are some contracts that contain debug logs using the `console.log` library from the hardhat, including:

- PolicyHelperV1
- StrategyLibV1

- AaveStrategy
- CompoundStrategy

Impact N/A

Suggestion Remove the debug logs.

Feedback from the Project We intend to keep the hardhat logs in the production environment as well for easy debugging and investigation on the mainnet, using Tenderly.

2.4 Additional Note

2.4.1 Potential centrality problems

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description Being served as an insurance project, the Neptune project relies on some external information to make decisions. As a result, certain centrality is introduced to let the project owners assign certain critical values or make decisions, including but not limited to:

- The cover content (specified by an IPFS URL) can be updated by invoking the `updateCover` function.
- There is a cool down period after each reporting in which the project owners are able to change the result of the report under emergency.
- The reward distribution in the `BondPool` and `StakingPool` relies on the project owners to transfer enough reward tokens to the contract, otherwise some of the users may not be able to withdraw the rewards.
- Any tokens left in the contracts which inherit from `Recoverable` can be extracted by the `RecoveryAgent`.
- The project owners are allowed to modify any storage values of the core `Store` contract, and add/remove contracts to/from the project.
- The governance is responsible for resolving and finalizing all incidence reports. Neither cover creators nor stakers/voters have the right to do that.

Impact N/A

Suggestion N/A