PEGO Network

Public Block Chain Security Audit

V1.0

No. 202305161441

May 16th, 2023

Contents

Overview

Project Overview

Project Name	PEGO Network
Audit Scope	https://github.com/pg-chain/pg-mainnet (PG Chain) https://github.com/pg-chain/pg-genesis-contract (PG Genesis Contract)
PG Chain Commit Hash	4458c44d8a4877ef58b09d2f73024b8810ac79eb (Initial) f2a41db58b53ace96850ffd8d51945e62a650a30 a8527ac997b6e40cf9e302f20cba1af33197ea74 eb74aed59e857da0e61bc8db282fcaba0527c5a7 74aa9e29b03b26a61527dc5df3846fb391dd4f58 d49314ddbd33cfa8b848e36895a664e981730763 7eafa75f3db2ec8ac478df19dfe7cd99f8efd8d0 12694416fb15f502173d4790ebfe13431d23e1b6 (Final)
PG Genesis Contract Commit Hash	9fafc3c13c9b69a4acf2be391e7b91aa429d1a0a (Initial) d748b7dc05299a1e8ba0e7a24e2db979786fd6a0 169278dd1aedeab73dfeba6198df1f243cdc6ce4 7c63b4b2c6944e8f6372c52e0159c2e6f43431ce 7441c4bb274ca53064e46139ecd10dc4ea13b400 f1c35b0afb768fc1db19770d2bcb60456de2241b b150a0e9ed5c68b7c530a31f11bece8e71b4e6b1 (Final)

Audit Overview

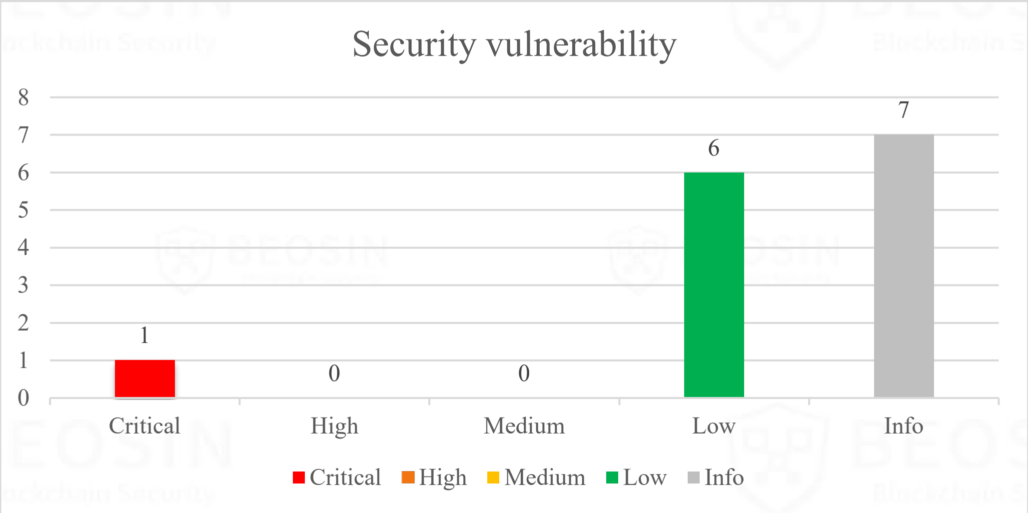
Audit work duration: April 21, 2023 – May 16, 2023

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team

Summary of audit results

After auditing, 1 Critical-risk, 6 Low-risk and 7 Info items were identified in the PEGO Network project. Specific audit details will be presented in the **Findings for Blockchain and Findings for Consensus Contracts** sections. Users should pay attention to the following aspects when interacting with this project:



Notes:

- **Notes for users:**

1. After users delegate their votes to a specific validator, the block rewards are not distributed to the users who voted for that validator, but rather to all users who have delegated their votes.
2. There is no emergency withdrawal function in the consensus contract, and users who deposit PGVT tokens and vote tokens may not be able to withdraw.

Findings for Blockchain

Index	Risk description	Severity level	Status
PG Chain-1	Error in LogInfo of *Parlia::distributeIncoming()* method	Info	Fixed

Finding Details:

[PG Chain-1] Error in LogInfo of *Parlia::distributeIncoming()* method

Severity Level Info

Lines consensus/parlia/parlia.go #L1294

Description (delegatorReward+header.GasUsed) is subtracted from the coinbaseRewards value recorded in logInfo [as shown in Figure 1], but the actual coinbaseRewards added in the state only subtracts the delegateReward, which is inconsistent with the actual situation. And when the coinbaseRewards reward is too small, a negative value will appear in the log. [as shown in Figure 2]

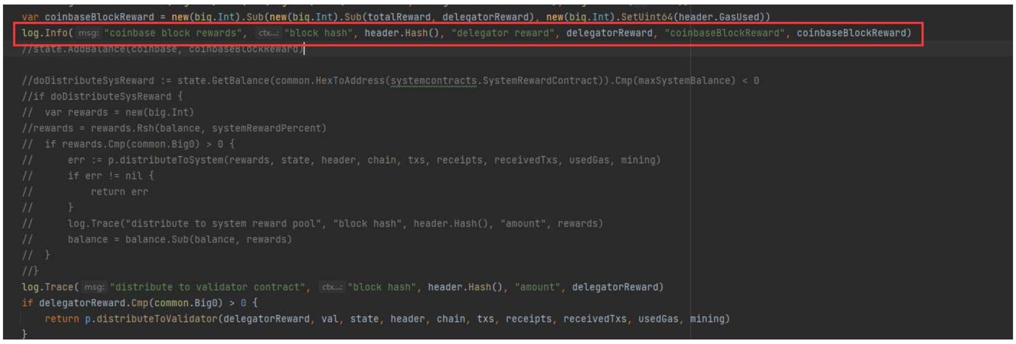


Figure 1 Source code of *distributeIncoming* method (unfixed)

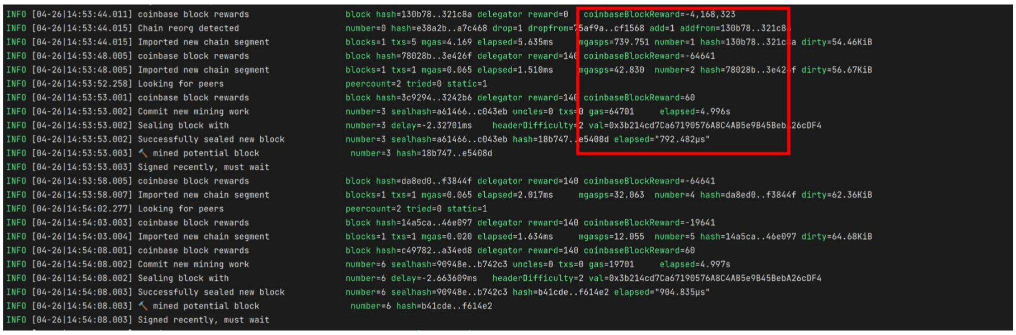


Figure 2 wrong log informations

Recommendations It is recommended to change the calculation method of coinbaseRewards recorded in the log.

Status Fixed. The log information has been removed.

Findings for Consensus Contracts

Index	Risk description	Severity level	Status
PG Genesis Contract-1	Price manipulation risk	Critical	Fixed
PG Genesis Contract-2	Incorrect balance checking logic	Low	Fixed
PG Genesis Contract-3	Precision conversion risk	Low	Fixed
PG Genesis Contract-4	Lack of emergency withdrawal function	Low	Acknowledged
PG Genesis Contract-5	The cool-down period can be bypassed	Low	Fixed
PG Genesis Contract-6	Potential reentrancy risk	Low	Fixed
PG Genesis Contract-7	Centralization risk	Low	Acknowledged
PG Genesis Contract-8	Unremoved test functions	Info	Fixed
PG Genesis Contract-9	Irregular code	Info	Fixed
PG Genesis Contract-10	Missing events	Info	Fixed
PG Genesis Contract-11	Comments error	Info	Fixed
PG Genesis Contract-12	The *getUnClaimedReward* function lacks the view keyword	Info	Fixed
PG Genesis Contract-13	Redundant codes	Info	Fixed

Status Notes:

- PG Genesis Contract-4 is acknowledged by PEGO project party, according to the project side, this is a normal logic, and the suggested scene will not appear, but users are advised to pay attention in real time.
- PG Genesis Contract-7 is acknowledged by PEGO project party, PEGO project party promises to give up permissions when it goes online.

Finding Details:

[PG Genesis Contract-1] Price manipulation risk

Severity Level	Critical
Type	Business Security
Lines	Pgvt.sol #L202-214

Description

In the Pgvt contract, the *stakingFarmingToken* function calculates the value of the staked LP tokens using the balance of wpg tokens inside the lpToken corresponding to the chef contract. If an attacker uses flash loans to deposit a large amount of wpg tokens into the LP pool, it will cause the pg-balance to be abnormally inflated and result in an unusually high amount of Pgvt tokens being minted.

```
193     function stakingFarmingToken(address token, uint256 amount) override external returns (bool) {
194         require(ftTokenAvailable[token],"token not support yet");
195         require(chef != address(0),"chef not config yet");
196         require(wpg != address(0),"wpg not config yet");
197         address user = msg.sender;
198         (bool success2) = IPEP20(token).transferFrom(user,address(this),amount);
199         require(success2, "transfer token failed");
200         lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
201 
202         address lpToken = IW3FT(chef).getTokenLp(token);
203         require(lpToken != address(0), "wrong");
204 
205         uint256 totalSupply = IPEP20(lpToken).totalSupply();
206 //          uint256 decimal = uint256(IPEP20(token).decimals());
207         require(totalSupply>0, "wrong");
208         uint256 pgbalance = IPEP20(wpg).balanceOf(lpToken);
209         require(pgbalance>=minLpPg, "wrong");
210         uint256 mintamount = pgbalance.mul(2).mul(amount).div(totalSupply);
211 
212         staking[token][user] = staking[token][user].add(amount);
213         stakingPgVt[token][user] = stakingPgVt[token][user].add(mintamount);
214         _mint(user,mintamount);
215 
216         return true;
217     }
```

Figure 3 Source code of *stakingFarmingToken* function (unfixed)

Recommendations	It is recommended to add a time-weighted algorithm to calculate the LP token value to ensure proposal safety.
Status	Fixed, a weighted algorithm oracle has been added to the source code to calculate prices.

```
function stakingFarmingToken(address token, uint256 amount) override external returns (bool) {
    require(ftTokenAvailable[token],"token not support yet");
    require(chef != address(0),"chef not config yet");
    require(wpg != address(0),"wpg not config yet");
    require(oracle != address(0),"oracle not config yet");
    address user = msg.sender;
    (bool success2) = IPEP20(token).transferFrom(user,address(this),amount);
    require(success2, "transfer token failed");
    lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);

    address lpToken = IW3FT(chef).getTokenLp(token);
    require(lpToken != address(0), "wrong");

    uint8 decimal = IPEP20(token).decimals();
    uint8 decimal1 = IPEP20(lpToken).decimals();
    require((decimal == decimal1) && (decimal == 18), "wrong decimal");

    uint256 pgbalance = IPEP20(wpg).balanceOf(lpToken);
    require(pgbalance>=minLpPg, "wrong");

    uint256 price = uint256(IOracle(oracle).consult(lpToken));
    uint256 mintamount = amount.mul(price).div(1e18);

    staking[token][user] = staking[token][user].add(amount);
    stakingPgVt[token][user] = stakingPgVt[token][user].add(mintamount);
    _mint(user,mintamount);

    return true;
}
```

Figure 4 Source code of_ *stakingFarmingToken* function (fixed)

[PG Genesis Contract-2] Incorrect balance checking logic

Severity Level	Low
Type	Business Security
Lines	Validator.sol #L142-143 DAOCoin.sol #L82

Description

In the validator contract, the *undelegate* function deducts the balance of the votes when checking if the remaining balance is sufficient. However, due to the fact that the vote function already deducts the balance during voting, this logic results in double deduction, and users cannot withdraw the remaining DAOCoin tokens if the number of tokens they voted for is greater than their remaining DAOCoin balance.

```
136  function undelegate(address validator, uint256 amount) override external noReentrant initParams {
137    require(amount > 0, "invalid undelegation amount");
138    require(block.timestamp >= pendingUndelegateTime[msg.sender][validator], "pending undelegation exist");
139
140    uint256 remainBalance = delegatedOfValidator[msg.sender][validator].sub(amount, "not enough funds");
141    address delegator = msg.sender;
142    uint256 balance = _balances[delegator];//super.balanceOf(delegator);
143    require(balance.sub(_votes[delegator]) >= amount,"not enough dao coin, maybe vote something");
144
145    uint256 value = candidateMap.get(validator);
146    require(value >= amount, "invalid candidate amount");
147    candidateMap.put(validator,value.sub(amount));
```

Figure 5 Source code of *undelegate* function (unfixed)

```
76    function vote(address account,uint256 amount) override external onlyGov returns (bool) {
77        require(account != address(0), "IPEP22: vote from the zero address");
78        require(_balances[account] >= amount, "IPEP22: no available balance");
79
80        _votes[account] = _votes[account].add(amount);
81        _balances[GOV_CONTRACT_ADDR] = _balances[GOV_CONTRACT_ADDR].add(amount);
82        _balances[account] = _balances[account].sub(amount);
83
84        return true;
85    }
```

Figure 6 Source code of *vote* function (unfixed)

Recommendations

It is recommended to only check if the current account balance is sufficient.

Status

Fixed, the source code modified the corresponding check.

```
136  function undelegate(address validator, uint256 amount) override external noReentrant initParams {
137    require(amount > 0, "invalid undelegation amount");
138    require(block.timestamp >= pendingUndelegateTime[msg.sender][validator], "pending undelegation exist");
139
140    uint256 remainBalance = delegatedOfValidator[msg.sender][validator].sub(amount, "not enough funds");
141    address delegator = msg.sender;
142    require(_balances[delegator] >= amount,"not enough dao coin, maybe vote something");
143
144    uint256 value = candidateMap.get(validator);
145    require(value >= amount, "invalid candidate amount");
146    candidateMap.put(validator,value.sub(amount));
147
```

Figure 7 Source code of *undelegate* function (fixed)

[PG Genesis Contract-3] Precision conversion risk

Severity Level Low

Type Business Security

Lines Pgvt.sol #L210

Description In the Pgvt contract, if a user stakes tokens with different decimal from PGVT using the *stakingFarmingToken* function, it will result in a discrepancy between the actual and expected amount of PGVT tokens earned.

```
193     function stakingFarmingToken(address token, uint256 amount) override external returns (bool) {
194         require(ftTokenAvailable[token],"token not support yet");
195         require(chef != address(0),"chef not config yet");
196         require(wpg != address(0),"wpg not config yet");
197         address user = msg.sender;
198         (bool success2) = IPEP20(token).transferFrom(user,address(this),amount);
199         require(success2, "transfer token failed");
200         lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
201
202         address lpToken = IW3FT(chef).getTokenLp(token);
203         require(lpToken != address(0), "wrong");
204
205         uint256 totalSupply = IPEP20(lpToken).totalSupply();
206 //          uint256 decimal = uint256(IPEP20(token).decimals());
207         require(totalSupply>0, "wrong");
208         uint256 pgbalance = IPEP20(wpg).balanceOf(lpToken);
209         require(pgbalance>=minLpPg, "wrong");
210         uint256 mintamount = pgbalance.mul(2).mul(amount).div(totalSupply);
211
212         staking[token][user] = staking[token][user].add(amount);
213         stakingPgVt[token][user] = stakingPgVt[token][user].add(mintamount);
214         _mint(user,mintamount);
215
216         return true;
217     }
```

Figure 8 Source code of *stakingFarmingToken* function (unfixed)

Recommendations It is recommended to use the converted amount with appropriate decimal for the calculation.

Status Fixed, source code added judgment for decimal.

```
197     function stakingFarmingToken(address token, uint256 amount) override external returns (bool) {
198         require(ftTokenAvailable[token],"token not support yet");
199         require(chef != address(0),"chef not config yet");
200         require(wpg != address(0),"wpg not config yet");
201         require(oracle != address(0),"oracle not config yet");
202         address user = msg.sender;
203         (bool success2) = IPEP20(token).transferFrom(user,address(this),amount);
204         require(success2, "transfer token failed");
205         lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
206
207         address lpToken = IW3FT(chef).getTokenLp(token);
208         require(lpToken != address(0), "wrong");
209
210         uint8 decimal = IPEP20(token).decimals();
211         uint8 decimal1 = IPEP20(lpToken).decimals();
212         require((decimal == decimal1) && (decimal == 18), "wrong decimal");
213
214         uint256 pgbalance = IPEP20(wpg).balanceOf(lpToken);
215         require(pgbalance>=minLpPg, "wrong");
216
217         uint256 price = uint256(IOracle(oracle).consult(lpToken));
218         uint256 mintamount = amount.mul(price).div(1e18);
219
220         staking[token][user] = staking[token][user].add(amount);
221         stakingPgVt[token][user] = stakingPgVt[token][user].add(mintamount);
222         _mint(user,mintamount);
223
224         return true;
225     }
```

Figure 9 Source code of *stakingFarmingToken* function (fixed)

[PG Genesis Contract-4] Lack of emergency withdrawal function

Severity Level	Low
Type	Business Security
Lines	Pgvt.sol #L175
Description	In the Pgvt contract, the *unStaking* function is unable to withdraw the remaining staked tokens if the user's _balances are insufficient, which may result in the user's staked tokens being locked inf certain situations.

```
172    function unStaking(address token) override external payable returns (bool) {
173        address user = msg.sender;
174        require(staking[token][user] > 0 ,"No enough funds");
175        require(stakingPgVt[token][user] <= _balances[user] ,"No enough pgvt");
176        require(block.timestamp >= lockTime[token][user] ,"Lock time existed");
177        lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
178        _burn(user, stakingPgVt[token][user]);
179        if (token == address(0)) {
180            (bool success1,) = msg.sender.call{value: staking[token][user]}("");
181            require(success1, "transfer pg failed");
182        } else {
183            (bool success2) = IPEP20(token).transfer(user,staking[token][user]);
184            require(success2, "transfer token failed");
185        }
186        staking[token][user] = 0;
187        stakingPgVt[token][user] = 0;
188
189        return true;
190    }
```

Figure 10 Source code of *unstake* function (unfixed)

Recommendations	It is recommended to add an emergency withdrawal function for staked tokens to be withdrawn in special circumstances.
Status	Acknowledged. The project party informed that it is related to business, and it is impossible to lock up.

[PG Genesis Contract-5] The cool-down period can be bypassed

Severity Level: Low

Type: Business Security

Lines: Validator.sol #L134,202-203

Description

In the validator contract, the inconsistency between the cool-down periods for staking and changing staking can lead to a bypass of the cool-down period for redeeming stakes in the *delegate* function. An attacker can use the *redelegate* function to move the stake to another validator and then use *undelegate* function to withdraw the corresponding stake without being subject to the cool-down period. This could become one of the attack chains used by attackers.

```
128     }
129     delegated[delegator] = delegated[delegator].add(amount);
130     delegatedOfValidator[delegator][validator] = delegatedOfValidator[delegator][validator].add(amount);
131 
132     delegatorDebt[delegator] = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION);
133 
134     pendingUndelegateTime[delegator][validator] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);
135 
136     _mint(delegator,amount);
```

Figure 11 Source code of *delegate* function(unfixed)

```
198     PoolInfo storage poolSrc = pool[validatorSrc];
199 
200     delegatedOfValidator[delegator][validatorSrc] = delegatedOfValidator[delegator][validatorSrc].sub(amount);
201     delegatedOfValidator[delegator][validatorDst] = delegatedOfValidator[delegator][validatorDst].add(amount);
202     pendingRedelegateTime[delegator][validatorSrc][validatorDst] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);
203     pendingRedelegateTime[delegator][validatorDst][validatorSrc] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);
```

Figure 12 Source code of *redelegate* function(unfixed)

Recommendations

It is recommended to synchronize the pendingUndelegateTime and pendingRedelegateTime variables into one variable.

Status

Fixed.

```
194 
195     delegatedOfValidator[delegator][validatorSrc] = delegatedOfValidator[delegator][validatorSrc].sub(amount);
196     delegatedOfValidator[delegator][validatorDst] = delegatedOfValidator[delegator][validatorDst].add(amount);
197     pendingUndelegateTime[delegator][validatorSrc] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);
198     pendingUndelegateTime[delegator][validatorDst] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);
199 
```

Figure 13 Source code of *redelegate* function(fixed)

[PG Genesis Contract-6] Potential reentrancy risk

Severity Level	Low
Type	Business Security
Lines	Validator.sol #L180-187
Description	The *Delegate*, *undelegate*, and *claimReward* functions in the Validator contract, and the *unStaking* function in the Pgvt contract, use the call function to transfer rewards or stakes to users. However, the function transfers occur before the ledger state changes, which poses a potential reentrancy risk.

```
172    function unStaking(address token) override external payable returns (bool) {
173        address user = msg.sender;
174        require(staking[token][user] > 0 ,"No enough funds");
175        require(stakingPgVt[token][user] <= _balances[user] ,"No enough pgvt");
176        require(block.timestamp >= lockTime[token][user] ,"Lock time existed");
177        lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
178        _burn(user, stakingPgVt[token][user]);
179        if (token == address(0)) {
180            (bool success1,) = msg.sender.call{value: staking[token][user]}("");
181            require(success1, "transfer pg failed");
182        } else {
183            (bool success2) = IPEP20(token).transfer(user,staking[token][user]);
184            require(success2, "transfer token failed");
185        }
186        staking[token][user] = 0;
187        stakingPgVt[token][user] = 0;
188
189        return true;
190    }
```

Figure 14 Source code of *unStaking* function (unfixed)

Recommendations	It is recommended to place the corresponding ledger state changes before the call invocation.
Status	Fixed.

```
function unStaking(address token) override external payable returns (bool) {
    address user = msg.sender;
    require(staking[token][user] > 0 ,"No enough funds");
    require(stakingPgVt[token][user] <= _balances[user] ,"No enough pgvt");
    require(block.timestamp >= lockTime[token][user] ,"Lock time existed");
    lockTime[token][user] = block.timestamp.add(lockTimeInMinutes * 1 minutes);
    _burn(user, stakingPgVt[token][user]);
    uint256 receiveAmount = staking[token][user];
    staking[token][user] = 0;
    stakingPgVt[token][user] = 0;
    if (token == address(0)) {
        (bool success1,) = msg.sender.call{value: receiveAmount}("");
        require(success1, "transfer pg failed");
    } else {
        (bool success2) = IPEP20(token).transfer(user,receiveAmount);
        require(success2, "transfer token failed");
    }
    return true;
}
```

Figure 15 Source code of *unStaking* function (fixed)

[PG Genesis Contract-7] Centralization risk

Severity Level	Low
Type	Business Security
Lines	Pgvt.sol #L240-247
Description	The configuration functions such as *setChef* and *setWpg* can be directly called by the administrator, which poses a centralization risk.

```
function setChef(address chef_)  external onlyOwner returns(bool) {
    chef = chef_ ;
    return true;
}

function setWpg(address wpg_)  external onlyOwner returns(bool) {
    wpg = wpg_ ;
    return true;
}
```

Figure 16 Source code of *setChef* and *setWpg* functions (unfixed)

Recommendations	It is recommended to use DAO governance or multi-signature wallets for management.
Status	Acknowledged. The project party promises to give up the authority after the launch is completed.

[PG Genesis Contract-8] Unremoved test functions

Severity Level	Info
Type	Coding Conventions
Lines	Validator.sol #L73-75 PnnNft.sol #L30-34
Description	The test functions such as *mintTo* and *mint* in the validator contract and PnnNft contract have not been deleted. If they are not deleted before deployment, they may be exploited by attackers.

```
73  function mintTo(address to, uint256 amount) external { //todo delete
74      _mint(to,amount);
75  }
```

Figure 17 Source code of *mintTo* function (unfixed)

```
30  function  mint(address to) external onlyOwner returns(uint256 tokenId) { // todo use for test
31      _mint(to,tokenIndex);
32      tokenId = tokenIndex;
33      tokenIndex++;
34  }
```

Figure 18 Source code of *mint* function (unfixed)

Recommendations	It is recommended to delete related test code.
Status	Fixed.

[PG Genesis Contract-9] Irregular codes

Severity Level Info

Type Coding Conventions

Lines DAOCoin.sol #L14-16

Description The visibility of _name, _symbol, and _decimals in the standard ERC20 contract should be private.

```
11  contract DAOCoin is IPEP22,IVT,System {
12      using SafeMath for uint256;
13
14      string public _name = "Vote Token";
15      string public _symbol = "VT";
16      uint8 public _decimals = 18;
17
```

Figure 19 Irregular codes(unfixed)

Recommendations It is recommended to modify the corresponding function visibility.

Status Fixed.

```
14      string internal _name = "Vote Token";
15      string internal _symbol = "VT";
16      uint8 internal _decimals = 18;
```

Figure 20 codes(fixed)

[PG Genesis Contract-10] Missing events

Severity Level	Info
Type	Coding Conventions
Lines	Pgvt.sol #L240-247
Description	Several functions that set critical parameters lack events.

```
    function setChef(address chef_)  external onlyOwner returns(bool) {
        chef = chef_ ;
        return true;
    }

    function setWpg(address wpg_)  external onlyOwner returns(bool) {
        wpg = wpg_ ;
        return true;
    }
```

Figure 21 Source code of *setChef* and *setWpg* functions (unfixed)

Recommendations	It is recommended to modify the corresponding function visibility.
Status	Fixed.

```
    function setChef(address chef_)  external onlyOwner returns(bool) {
        chef = chef_ ;
        emit putChef(chef);
        return true;
    }

    function setWpg(address wpg_)  external onlyOwner returns(bool) {
        wpg = wpg_ ;
        emit putWpg(wpg);
        return true;
    }
```

Figure 22 Source code of *setChef* and *setWpg* functions (fixed)

[PG Genesis Contract-11] Comments error

Severity Level Info

Type Coding Conventions

Lines SystemDAO.sol#L21

Description In the SystemDAO contract, the value of the MAX_MINT_PG variable is 1e24 and cannot be changed, which is inconsistent with the comment that states 1e25.

```
19      mapping(uint256 => uint256) pgMintAmount; //PG mint amount record , heigh
20
21      uint256 public constant MAX_MINT_PG = 1e24;//max mint pg limit 1e25
22      uint256 public constant MAX_DELEGATOR_REWARD_PERCENT = 10000;// 100%
23      uint256 public constant MIN_DELEGATOR_REWARD_PERCENT = 1000;// 10%
24      uint256 public constant DEFAULT_BLOCK_REWARD = 0;
25
26
```

Figure 23 MAX_MINT_PG comments(unfixed)

Recommendations It is recommended to change the comments.

Status Fixed.

```
21      uint256 public constant MAX_MINT_PG = 1e24;//max mint pg limit 1e24
22      uint256 public constant MAX_DELEGATOR_REWARD_PERCENT = 10000;// 100%
23      uint256 public constant MIN_DELEGATOR_REWARD_PERCENT = 1000;// 10%
24      uint256 public constant DEFAULT_BLOCK_REWARD = 0;
```

Figure 24 MAX_MINT_PG comments(fixed)

[PG Genesis Contract-12] The *getUnClaimedReward* function lacks the view keyword

Severity Level	Info
Type	Coding Conventions
Lines	Validator.sol#L277-279

Description

In the Validator contract, the *getUnClaimedReward* function only serves a query purpose but is not marked with the "view" keyword, which can result in unnecessary gas consumption when called.

```
277 function getUnClaimedReward(address delegator) override external returns(uint256 amount) {
278     amount = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
279 }
```

Figure 25 Source code of *getUnClaimedReward* function (unfixed)

Recommendations

It is recommended to add the view keyword.

Status

Fixed.

```
function getUnClaimedReward(address delegator) override external view returns(uint256 amount) {
    amount = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
}
```

Figure 26 Source code of *getUnClaimedReward* function (unfixed)

[PG Genesis Contract-13] Redundant codes

Severity Level	Info
Type	Coding Conventions
Lines	System.sol #L72-74 Validator.sol #L103,119,141,154,277-279,297-299
Description	In the system contract, the code for the *onlyDaoContract* and *onlySysDao* modifiers is duplicated, and they are not used in other contracts.

```
61  modifier onlySysDao() {
62        require(msg.sender == SYSTEM_DAO_ADDR, "the message sender must be system dao contract");
63    _;
64  }
65
66  modifier onlyValidatorContract() {
67    require(msg.sender == VALIDATOR_CONTRACT_ADDR, "the message sender must be validatorSet contract");
68    _;
69  }
70
71
72  modifier onlyDaoContract() {
73    require(msg.sender == SYSTEM_DAO_ADDR, "the message sender must be dao contract");
74    _;
75  }
76
```

Figure 27 Source code of *onlyDaofcontract* modifier(unfixed)

In the Validator contract, when using the *delegate* and *undelegate* functions, users only use PGVT tokens for staking and withdrawal, but these functions have unnecessary payable tags.

```
102   /****************************** External functions ******************************/
103   function delegate(address validator, uint256 amount) override external noReentrant initParams payable {
104     require(amount > 0, "invalid delegate amount");
105
106     bool suc = IPEP20(PGVT_CONTRACT_ADDR).transferFrom(msg.sender,address(this),amount);
107     require(suc, "transferFrom call fail");
108
109     address delegator = msg.sender;
110
111     uint256 value = candidateMap.get(validator);
112     if (value>0) {
113       candidateMap.put(validator,value.add(amount));
114     } else {
115       candidateMap.put(validator,amount);
116     }
117
```

Figure 28 Source code of *delegate* function(unfixed)

```
141   function undelegate(address validator, uint256 amount) override external noReentrant initParams payable {
142     require(amount > 0, "invalid undelegation amount");
143     require(block.timestamp >= pendingUndelegateTime[msg.sender][validator], "pending undelegation exist");
144
145     uint256 remainBalance = delegatedOfValidator[msg.sender][validator].sub(amount, "not enough funds");
146     address delegator = msg.sender;
147     uint256 balance = _balances[delegator];//super.balanceOf(delegator);
148     require(balance.sub(_votes[delegator]) >= amount,"not enough dao coin, maybe vote something");
149
150     uint256 value = candidateMap.get(validator);
151     require(value >= amount, "invalid candidate amount");
152     candidateMap.put(validator,value.sub(amount));
153
```

Figure 29 Source code of *delegate* function(unfixed)

In the Validator contract, the *delegate* and *undelegate* functions retrieve the pool information for the validator but do not use it.

```
function delegate(address validator, uint256 amount) override external noReentrant
  require(amount > 0, "invalid delegate amount");

  bool suc = IPEP20(PGVT_CONTRACT_ADDR).transferFrom(msg.sender,address(this),amount);
  require(suc, "transferFrom call fail");

  address delegator = msg.sender;

  uint256 value = candidateMap.get(validator);
  if (value>0) {
    candidateMap.put(validator,value.add(amount));
  } else {
    candidateMap.put(validator,amount);
  }

  esurePool(validator);
  PoolInfo storage vpool = pool[validator];

```

Figure 30 Source code of *delegate* function(unfixed)

```
function undelegate(address validator, uint256 amount) override external noReentrant initParams payable {
  require(amount > 0, "invalid undelegation amount");
  require(block.timestamp >= pendingUndelegateTime[msg.sender][validator], "pending undelegation exist");

  uint256 remainBalance = delegatedOfValidator[msg.sender][validator].sub(amount, "not enough funds");
  address delegator = msg.sender;
  uint256 balance = _balances[delegator];//super.balanceOf(delegator);
  require(balance.sub(_votes[delegator]) >= amount,"not enough dao coin, maybe vote something");

  uint256 value = candidateMap.get(validator);
  require(value >= amount, "invalid candidate amount");
  candidateMap.put(validator,value.sub(amount));

  PoolInfo storage vpool = pool[validator];

  if (accPgPerShare > 0 && delegated[delegator] > 0) {
    //将之前奖励打给用户
    uint256 pending = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
    if (pending > 0) {
      (bool success,) = msg.sender.call{value: pending}("");
      require(success, "transfer rewards failed");
    }
  }
```

Figure 31 Source code of *delegate* function(unfixed)

In the Validator contract, the *getUnClaimedReward* and *getDistributedReward* functions have the same parameters, functionality, and return values.

```
function getUnClaimedReward(address delegator) override external returns(uint256 amount) {
  amount = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
}

function getDelegated(address delegator, address validator) override external view returns(uint256) {
  return delegatedOfValidator[delegator][validator];
}

function getValidatorDelegated(address validator) override external view returns(uint256) {
  return candidateMap.get(validator);
}

function getValidatorEntries(uint256 k) external view returns( address[] memory keys) {
  keys = candidateMap.getTopKeys(k);
}

function getTotalDelegated(address delegator) override external view returns(uint256) {
  return delegated[delegator];
}

function getDistributedReward(address delegator) override external view returns(uint256) {
  return delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
}
```

Figure 32 Source code of *getUnClaimedReward* and *getDistributedReward* functions(unfixed)

Recommendations	It is recommended to delete related redundant code.
Status	Fixed.

Blockchain Audit Contents

The PEGO is a modular and extensible framework for building Ethereum-compatible blockchain networks and general scaling solutions.

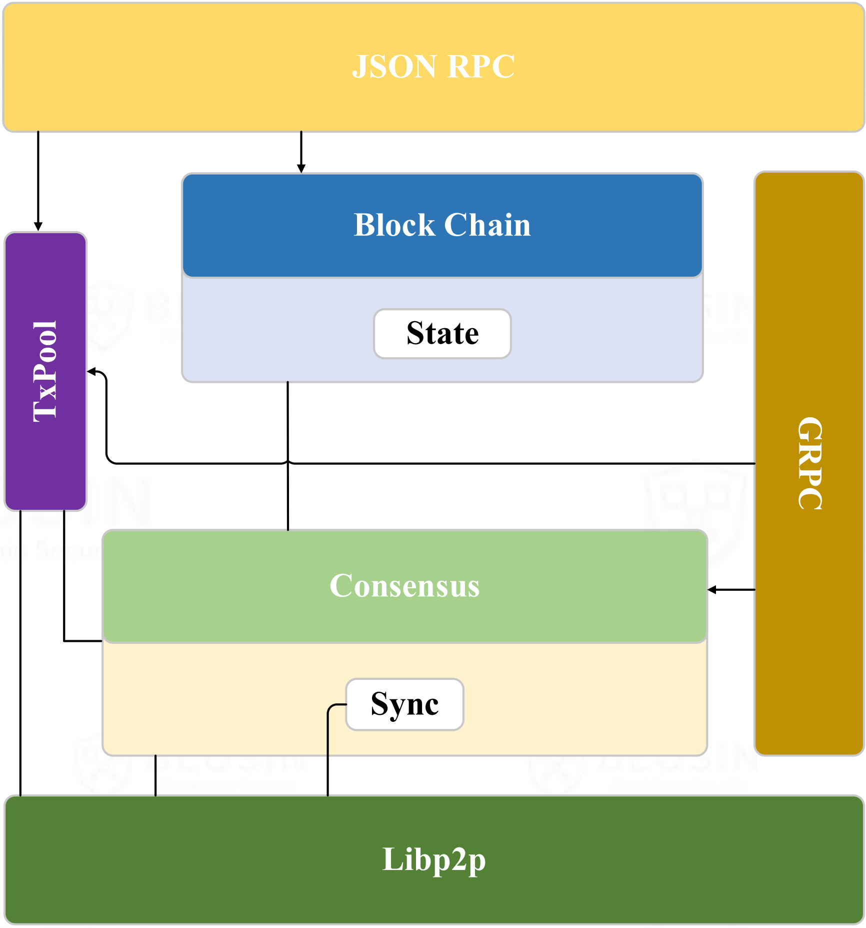


Figure 33 The PEGO overall framework

1 JSON RPC Security Audit

1.1 JSON RPC Introduction

RPC provides a way to access the exported objects through the grid or other I/O connections. RPC serializes the methods and parameters to be called and transmits them to the other side through TCP/UDP protocol, and the other side deserializes them to find the methods and parameters to be called, and then serializes the return values to return. After creating the RPC service, the object can be registered to the service to make it accessible to the outside. The export method according to this specific way is called remote call, which also supports the publish/subscribe model.

When the PEGO node starts, it will pull up the JSON RPC service, register the corresponding RPC module, and use RPC or HTTP to call the corresponding interface. The JSON RPC implemented by PEGO node is basically the same as the RPC module commonly used in Ethereum, which includes four methods: *GetSnapshot, GetSnapshotAtHash, GetValidators* and *GetValidatorsAtHash.*

```go
// GetSnapshot retrieves the state snapshot at a given block.
func (api *API) GetSnapshot(number *rpc.BlockNumber) (*Snapshot, error) {
    // Retrieve the requested block number (or current if none requested)
    var header *types.Header
    if number == nil || *number == rpc.LatestBlockNumber {
        header = api.chain.CurrentHeader()
    } else {
        header = api.chain.GetHeaderByNumber(uint64(number.Int64()))
    }
    // Ensure we have an actually valid block and return its snapshot
    if header == nil {
        return nil, errUnknownBlock
    }
    return api.parlia.snapshot(api.chain, header.Number.Uint64(), header.Hash(), nil)
}

// GetSnapshotAtHash retrieves the state snapshot at a given block.
func (api *API) GetSnapshotAtHash(hash common.Hash) (*Snapshot, error) {
    header := api.chain.GetHeaderByHash(hash)
    if header == nil {
        return nil, errUnknownBlock
    }
    return api.parlia.snapshot(api.chain, header.Number.Uint64(), header.Hash(), nil)
}

// GetValidators retrieves the list of validators at the specified block.
func (api *API) GetValidators(number *rpc.BlockNumber) ([]common.Address, error) {
    // Retrieve the requested block number (or current if none requested)
    var header *types.Header
    if number == nil || *number == rpc.LatestBlockNumber {
        header = api.chain.CurrentHeader()
    } else {
        header = api.chain.GetHeaderByNumber(uint64(number.Int64()))
    }
    // Ensure we have an actually valid block and return the validators from its snapshot
    if header == nil {
        return nil, errUnknownBlock
    }
    snap, err := api.parlia.snapshot(api.chain, header.Number.Uint64(), header.Hash(), nil)
    if err != nil {
        return nil, err
    }
    return snap.validators(), nil
}
```

Figure 34 JSON RPC consortium module(1/2)

```go
// GetValidatorsAtHash retrieves the list of validators at the specified block.
func (api *API) GetValidatorsAtHash(hash common.Hash) ([]common.Address, error) {
    header := api.chain.GetHeaderByHash(hash)
    if header == nil {
        return nil, errUnknownBlock
    }
    snap, err := api.parlia.snapshot(api.chain, header.Number.Uint64(), header.Hash(), nil)
    if err != nil {
        return nil, err
    }
    return snap.validators(), nil
}
```

Figure 35 JSON RPC consortium module(2/2)

The consensus mechanism used in this audit contains four interfaces: *GetSnapshot, GetSnapshotAtHash and GetValidators, GetValidatorsAtHash.*

1. GetSnapshot: This interface is used to retrieve the snapshot information of the current BSC network. Snapshot refers to the state information of all accounts in the current network, including account balances, contract codes, transaction records, etc. By calling the GetSnapshot interface, you can obtain the snapshot information of the current network and analyze and process it as needed.

2. GetSnapshotAtHash: This interface is used to retrieve the snapshot information corresponding to a specified

block hash value. Unlike the GetSnapshot interface, the GetSnapshotAtHash interface can retrieve the corresponding snapshot information based on a specified block hash value instead of the current network's snapshot information. This function can be used to query the state information of a specific block.

3. GetValidators: This interface is used to retrieve the validator list in the current BSC network. Validators refer to nodes that participate in verifying transactions and creating new blocks. By calling the GetValidators interface, users can obtain a list of all validator nodes in the current network and understand the distribution of validators in the network.

4. GetValidatorsAtHash: This interface is used to retrieve the validator list corresponding to a specified block hash value. Similar to the GetValidators interface, the GetValidatorsAtHash interface can retrieve the corresponding validator list based on a specified block hash value instead of the current validator list in the network. This function can be used to query validator node information at a specific block time.

1.2 JSON RPC Processing Logic

In PEGO node, after receiving an RPC request, the node will call the *serveSingleRequest* function to check the request function type, to parse the corresponding RPC request and locate the corresponding module and function to call.

```go
// serveSingleRequest reads and processes a single RPC request from the given codec. This
// is used to serve HTTP connections. Subscriptions and reverse calls are not allowed in
// this mode.
func (s *Server) serveSingleRequest(ctx context.Context, codec ServerCodec) {
	// Don't serve if server is stopped.
	if atomic.LoadInt32(&s.run) == 0 {
		return
	}

	h := newHandler(ctx, codec, s.idgen, &s.services)
	h.allowSubscribe = false
	defer h.close(io.EOF, nil)

	reqs, batch, err := codec.readBatch()
	if err != nil {
		if err != io.EOF {
			codec.writeJSON(ctx, errorMessage(&invalidMessageError{"parse error"}))
		}
		return
	}
	if batch {
		h.handleBatch(reqs)
	} else {
		h.handleMsg(reqs[0])
	}
}
```

Figure 36 Source code of *serveSingleRequest* function

1.3 RPC Sensitive Interface Permission

The vast majority of the RPC interfaces currently provided by PEGO node are for querying data, and there are no high authority interfaces.

1.4 CLI Commands Security Audit

The PEGO's CLI is mainly for nodes to perform related configurations, and its corresponding parameters are as follows:

```
COMMANDS:
   account                          Manage accounts
   attach                           Start an interactive JavaScript environment (connect to node)
   console                          Start an interactive JavaScript environment
   db                               Low level database operations
   dump                             Dump a specific block from storage
   dumpconfig                       Show configuration values
   dumpgenesis                      Dumps genesis block JSON configuration to stdout
   export                           Export blockchain into file
   export-preimages                 Export the preimage database into an RLP stream
   import                           Import a blockchain file
   import-preimages                 Import the preimage database from an RLP stream
   init                             Bootstrap and initialize a new genesis block
   js                               Execute the specified JavaScript files
   license                          Display license information
   makecache                        Generate ethash verification cache (for testing)
   makedag                          Generate ethash mining DAG (for testing)
   removedb                         Remove blockchain and state databases
   show-deprecated-flags            Show flags that have been deprecated
   snapshot                         A set of commands based on the snapshot
   version                          Print version numbers
   version-check                    Checks (online) whether the current version suffers from any known security vulnerabilities
   wallet                           Manage Ethereum presale wallets
   help, h                          Shows a list of commands or help for one command
```

Figure 37 The PEGO node command line parameters

After testing, the relevant commands in the CLI meet the audit requirements and there is no security risk.

1.5 Node Account Unlocking Security Audit

PEGO uses the DPoS consensus. The DPoS consensus requires that the coinbase account of the node must be unlocked after it is activated, otherwise it will not be able to generate blocks. There is a potential risk here, that is, if the node opens the RPC interface to the public network, any user connected to the node can operate the node's coinbase, which may cause the node to crash when running the consortium V1 consensus. But according to the project party, this interface is only open to 127.0.0.1.

In theory, unlocked accounts have the above-mentioned risks on open JSON-RPC nodes (users cannot confirm whether their API is used by hackers to sign raw data, unless the log is turned on). Therefore, it is recommended to close the sensitive RPC or only open it to the whitelist addresses when the node is online.

2 Node Security

2.1 Number of Node Connections

The PEGO nodes can use the networking module to set parameters at startup, where --maxpeers limit the number of links to the node, and if the number of links exceeds the specified value, no links will be made.

```
NETWORKING OPTIONS:
  --bootnodes value           Comma separated enode URLs for P2P discovery bootstrap
  --discovery.dns value       Sets DNS discovery entry points (use "" to disable DNS)
  --port value                Network listening port (default: 30303)
  --maxpeers value            Maximum number of network peers (network disabled if set to 0) (default: 50)
  --maxpendpeers value        Maximum number of pending connection attempts (defaults used if set to 0) (default: 0)
  --nat value                 NAT port mapping mechanism (any|none|upnp|pmp|extip:<IP>) (default: "any")
  --nodiscover                Disables the peer discovery mechanism (manual peer addition)
  --v5disc                    Enables the experimental RLPx V5 (Topic Discovery) mechanism
  --netrestrict value         Restricts network communication to the given IP networks (CIDR masks)
  --nodekey value             P2P node key file
  --nodekeyhex value          P2P node key as hex (for testing)
```

Figure 38 Node start-up parameters

After testing, the PEGO nodes can limit the number of connections to the current node using the server parameter (which defaults even if not specified), which can prevent the node from having too many connections.

```
TRACE[02-24|16:24:15.042] Found seed node in database    id=4ae70c9793b68453 addr=127.0.0.1:30314 age=1m36.0426578s
TRACE[02-24|16:24:15.042] Found seed node in database    id=68934d9ff5e18b54 addr=127.0.0.1:30315 age=1m52.0426578s
TRACE[02-24|16:24:15.042] Found seed node in database    id=822150deaaa449a1 addr=127.0.0.1:30310 age=5m38.0426578s
TRACE[02-24|16:24:15.042] Failed p2p handshake           id=4ae70c9793b68453 addr=127.0.0.1:30314 conn=dyndial err="too many peers"
TRACE[02-24|16:24:15.042] Found seed node in database    id=822150deaaa449a1 addr=127.0.0.1:30310 age=5m38.0426578s
```

Figure 39 Node connection information

2.2 Packet Size Limit

As shown in the figure below, the size of the data is checked when a transaction is received, which can avoid DoS attacks on nodes.

```go
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
	// Accept only legacy transactions until EIP-2718/2930 activates.
	if !pool.eip2718 && tx.Type() != types.LegacyTxType {
		return ErrTxTypeNotSupported
	}
	// Reject dynamic fee transactions until EIP-1559 activates.
	if !pool.eip1559 && tx.Type() == types.DynamicFeeTxType {
		return ErrTxTypeNotSupported
	}
	// Reject transactions over defined size to prevent DOS attacks
	if uint64(tx.Size()) > txMaxSize {
		return ErrOversizedData
	}
	// Transactions can't be negative. This may never happen using RLP decoded
	// transactions but may occur if you create a transaction using the RPC.
	if tx.Value().Sign() < 0 {
		return ErrNegativeValue
	}
	// Ensure the transaction doesn't exceed the current block limit gas.
	if pool.currentMaxGas < tx.Gas() {
		return ErrGasLimit
	}
	// Sanity check for extremely large numbers
	if tx.GasFeeCap().BitLen() > 256 {
		return ErrFeeCapVeryHigh
	}
	if tx.GasTipCap().BitLen() > 256 {
		return ErrTipVeryHigh
	}
	// Ensure gasFeeCap is greater than or equal to gasTipCap.
	if tx.GasFeeCapIntCmp(tx.GasTipCap()) < 0 {
		return ErrTipAboveFeeCap
	}
	// Make sure the transaction is signed properly.
	from, err := types.Sender(pool.signer, tx)
	if err != nil {
		return ErrInvalidSender
	}
	// Drop non-local transactions under our own minimal accepted gas price or tip
	if !local && tx.GasTipCapIntCmp(pool.gasPrice) < 0 {
		return ErrUnderpriced
	}
```

Figure 40 Source code for the *validateTX* function(1/2)

```go
	// Ensure the transaction adheres to nonce ordering
	if pool.currentState.GetNonce(from) > tx.Nonce() {
		return ErrNonceTooLow
	}
	// Transactor should have enough funds to cover the costs
	// cost == V + GP * GL
	if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
		return ErrInsufficientFunds
	}
	// Ensure the transaction has more gas than the basic tx fee.
	intrGas, err := IntrinsicGas(tx.Data(), tx.AccessList(), tx.To() == nil, true, pool.istanbul)
	if err != nil {
		return err
	}
	if tx.Gas() < intrGas {
		return ErrIntrinsicGas
	}
	return nil
}
```

Figure 41 Source code for the *validateTX* function(2/2)

2.3 Node Network Access Restrictions

Network parameters can be set when the node starts the chain, which can protect the node from attacks to some extent.

```
NETWORKING OPTIONS:
  --bootnodes value                   Comma separated enode URLs for P2P discovery bootstrap
  --discovery.dns value               Sets DNS discovery entry points (use "" to disable DNS)
  --port value                        Network listening port (default: 30303)
  --maxpeers value                    Maximum number of network peers (network disabled if set to 0) (default: 50)
  --maxpendpeers value                Maximum number of pending connection attempts (defaults used if set to 0) (default: 0)
  --nat value                         NAT port mapping mechanism (any|none|upnp|pmp|extip:<IP>) (default: "any")
  --nodiscover                        Disables the peer discovery mechanism (manual peer addition)
  --v5disc                            Enables the experimental RLPx V5 (Topic Discovery) mechanism
  --netrestrict value                 Restricts network communication to the given IP networks (CIDR masks)
  --nodekey value                     P2P node key file
  --nodekeyhex value                  P2P node key as hex (for testing)
```

Figure 42 The networking module parameters

3 Account &Asset Security

3.1 Account Model

The PEGO node uses a same account model with Ethereum, with the following basic data structure:

```
// Account is a modified version of a state.Account, where the root is replaced
// with a byte slice. This format can be used to represent full-consensus format
// or slim-snapshot format which replaces the empty root and code hash as nil
// byte slice.
type Account struct {
    Nonce    uint64
    Balance  *big.Int
    Root     []byte
    CodeHash []byte
}
```

Figure 43 Source code of the data structure of Account

- Nonce: The serial number of the transaction sent by the account;
- Balance: The balance of the account's platform coins;
- Root: The root hash []byte of the storage state tree;
- CodeHash: The EVM code bound to the account;

3.2 Account Generation

(1) externally owned account

The PEGO use Ethereum way to create external accounts

Through the RPC module personal to create an account interface: NewAccount

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
    _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
    if err != nil {
        return accounts.Account{}, err
    }
    // Add the account to the cache immediately rather
    // than waiting for file system notifications to pick it up.
    ks.cache.add(account)
    ks.refreshWallets()
    return account, nil
}
```

Figure 44 Source code of the *NewAccount* function

This interface will call the NewAccount function in accounts/keystore/keystore.go to generate an account and return the account address.

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
    _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
    if err != nil {
        return accounts.Account{}, err
    }
    // Add the account to the cache immediately rather
    // than waiting for file system notifications to pick it up.
    ks.cache.add(account)
    ks.refreshWallets()
    return account, nil
}
```

Figure 45 Source code of the *NewAccount* function

The NewAccount function in keystore.go will actually call the storeNewKey function in the accounts/keystore/key.go file to create an account, generate a public and private key, and store the created account in the cache.

```
func newKey(rand io.Reader) (*Key, error) {
    privateKeyECDSA, err := ecdsa.GenerateKey(crypto.S256(), rand)
    if err != nil {
        return nil, err
    }
    return newKeyFromECDSA(privateKeyECDSA), nil
}

func storeNewKey(ks keyStore, rand io.Reader, auth string) (*Key, accounts.Account, error) {
    key, err := newKey(rand)
    if err != nil {
        return nil, accounts.Account{}, err
    }
    a := accounts.Account{
        Address: key.Address,
        URL:     accounts.URL{Scheme: KeyStoreScheme, Path: ks.JoinPath(keyFileName(key.Addres
    }
    if err := ks.StoreKey(a.URL.Path, key, auth); err != nil {
        zeroKey(key.PrivateKey)
        return nil, a, err
    }
    return key, a, err
}
```

Figure 46 Source code of the *storeNewKey* function and *newKey* function

The storeNewKey function calls the newKey function to generate a complete key instance (including the account public and private key and account address), and calls the StoreKey function to write the generated account file to the local for persistent storage.

```go
// Generate an elliptic curve public / private keypair. If params is nil,
// the recommended default parameters for the key will be chosen.
func GenerateKey(rand io.Reader, curve elliptic.Curve, params *ECIESParams) (prv *PrivateKey, err error) {
	pb, x, y, err := elliptic.GenerateKey(curve, rand)
	if err != nil {
		return
	}
	prv = new(PrivateKey)
	prv.PublicKey.X = x
	prv.PublicKey.Y = y
	prv.PublicKey.Curve = curve
	prv.D = new(big.Int).SetBytes(pb)
	if params == nil {
		params = ParamsFromCurve(curve)
	}
	prv.PublicKey.Params = params
	return
}
```

Figure 47 Source code of the *GenerateKey* function

The *newKey* function will call *GenerateKey* function to generate the public and private keys.

```go
func newKeyFromECDSA(privateKeyECDSA *ecdsa.PrivateKey) *Key {
	id, err := uuid.NewRandom()
	if err != nil {
		panic(fmt.Sprintf("Could not create random uuid: %v", err))
	}
	key := &Key{
		Id:         id,
		Address:    crypto.PubkeyToAddress(privateKeyECDSA.PublicKey),
		PrivateKey: privateKeyECDSA,
	}
	return key
}
```

Figure 48 Source code of the *GnewkeyFromECDSA* function

Then pass the private key into the *newKeyFromECDSA* function, and then call the *PubkeyToAddress* function in the crypto/crypto.go file to calculate the account address according to the public key, thus generating a complete key instance.

```go
func PubkeyToAddress(p ecdsa.PublicKey) common.Address {
	pubBytes := FromECDSAPub(&p)
	return common.BytesToAddress(Keccak256(pubBytes[1:])[12:])
}
```

Figure 49 Source code of the *PubkeyToAddress* function

(2) contract account

PEGO node supports Ethereum Create and Create2 instructions to create contract accounts.

```go
// Create creates a new contract using code as deployment code.
func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int) (ret []byte, contractAddr common.Address, leftOverGas uint64, err error) {
    contractAddr = crypto.CreateAddress(caller.Address(), evm.StateDB.GetNonce(caller.Address()))
    return evm.create(caller, &codeAndHash{code: code}, gas, value, contractAddr, CREATE)
}

// Create2 creates a new contract using code as deployment code.
//
// The different between Create2 with Create is Create2 uses sha3(0xff ++ msg.sender ++ salt ++ sha3(init_code))[12:]
// instead of the usual sender-and-nonce-hash as the address where the contract is initialized at.
func (evm *EVM) Create2(caller ContractRef, code []byte, gas uint64, endowment *big.Int, salt *uint256.Int) (ret []byte, contractAddr common.Address, leftOverGas uint64, err error) {
    codeAndHash := &codeAndHash{code: code}
    contractAddr = crypto.CreateAddress2(caller.Address(), salt.Bytes32(), codeAndHash.Hash().Bytes())
    return evm.create(caller, codeAndHash, gas, endowment, contractAddr, CREATE2)
}
```

Figure 50 Source code of the *Create* function and *Create2* function

3.3 Transaction Signature

First call the *SignTx* function in the accounts/keystore/keystore/wallet.go file to check whether the account is unlocked.

```go
// SignTx implements accounts.Wallet, attempting to sign the given transaction
// with the given account. If the wallet does not wrap this particular account,
// an error is returned to avoid account leakage (even though in theory we may
// be able to sign via our shared keystore backend).
func (w *keystoreWallet) SignTx(account accounts.Account, tx *types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    // Make sure the requested account is contained within
    if !w.Contains(account) {
        return nil, accounts.ErrUnknownAccount
    }
    // Account seems valid, request the keystore to sign
    return w.keystore.SignTx(account, tx, chainID)
}
```

Figure 51 Source code of the *SignTx* function

If it is confirmed that the account is valid, call the *SignTx* function in the accounts/keystore/keystore.go file to sign.

```go
// SignTx signs the given transaction with the requested account.
func (ks *KeyStore) SignTx(a accounts.Account, tx *types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    // Look up the key to sign with and abort if it cannot be found
    ks.mu.RLock()
    defer ks.mu.RUnlock()

    unlockedKey, found := ks.unlocked[a.Address]
    if !found {
        return nil, ErrLocked
    }
    // Depending on the presence of the chain ID, sign with 2718 or homestead
    signer := types.LatestSignerForChainID(chainID)
    return types.SignTx(tx, signer, unlockedKey.PrivateKey)
}
```

Figure 52 Source code of the *SignTx* function

Because the chainID is not empty, in order to prevent repeated attacks across chains, EIP155Signer is selected as the signer and the *SignTx* function in core/types/transaction_signing.go is called to complete the transaction signature.

```
// SignTx signs the transaction using the given signer and private key.
func SignTx(tx *Transaction, s Signer, prv *ecdsa.PrivateKey) (*Transaction, error) {
    h := s.Hash(tx)
    sig, err := crypto.Sign(h[:], prv)
    if err != nil {
        return nil, err
    }
    return tx.WithSignature(s, sig)
}
```

Figure 53 Source code of the *SignTx* function

```
func NewEIP155Signer(chainId *big.Int) EIP155Signer {
    if chainId == nil {
        chainId = new(big.Int)
    }
    return EIP155Signer{
        chainId:    chainId,
        chainIdMul: new(big.Int).Mul(chainId, big.NewInt(2)),
    }
}
```

Figure 54 Source code of the *NewEIP155Signer* function

3.4 Asset Security

(1) Platform Asset Security

After a comprehensive audit, it can be confirmed that when running the DPoS (consortium V2) consensus, no new PEGO native tokens will be generated when validator nodes package blocks (call *FinalizeAndAssemble* function). This means that all native tokens have been fully pre-mined, and no new native tokens will be generated afterwards.

```go
func (c *Consortium) FinalizeAndAssemble (chain consensus.ChainHeaderReader, header *types.Header, state *state.StateDB,
    txs []*types.Transaction, uncles []*types.Header, receipts []*types.Receipt) (*types.Block, []*types.Receipt, error) {
    if err := c.initContract(); err != nil {
        return nil, nil, err
    }

    // No block rewards in PoA, so the state remains as is and uncles are dropped
    if txs == nil {
        txs = make([]*types.Transaction, 0)
    }
    if receipts == nil {
        receipts = make([]*types.Receipt, 0)
    }
    evmContext := core.NewEVMBlockContext(header, consortiumCommon.ChainContext{Chain: chain, Consortium: c}, &header.Coinbase, chain.OpEvents()...)
    transactOpts := &consortiumCommon.ApplyTransactOpts{
        ApplyMessageOpts: &consortiumCommon.ApplyMessageOpts{
            State:       state,
            Header:      header,
            ChainConfig: c.chainConfig,
            EVMContext:  &evmContext,
        },
        Txs:      &txs,
        Receipts: &receipts,
        // a.k.a. System Txs
        // It always equals nil since FinalizeAndAssemble doesn't receive any transactions
        ReceivedTxs: nil,
        UsedGas:     &header.GasUsed,
        Mining:      true,
        Signer:      c.signer,
        SignTxFn:    c.signTxFn,
    }

    if err := c.processSystemTransactions(chain, header, transactOpts, true); err != nil {
        return nil, nil, err
    }

    // should not happen. Once happen, stop the node is better than broadcast the block
    if header.GasLimit < header.GasUsed {
        return nil, nil, errors.New("gas consumption of system txs exceed the gas limit")
    }
    header.UncleHash = types.CalcUncleHash(nil)
    var blk *types.Block
    var rootHash common.Hash
    wg := sync.WaitGroup{}
    wg.Add(2)
    go func() {
        rootHash = state.IntermediateRoot(chain.Config().IsEIP158(header.Number))
        wg.Done()
    }()
    go func() {
        blk = types.NewBlock(header, *transactOpts.Txs, nil, *transactOpts.Receipts, trie.NewStackTrie(nil))
        wg.Done()
    }()
    wg.Wait()
    blk.SetRoot(rootHash)
    // Assemble and return the final block for sealing
    return blk, *transactOpts.Receipts, nil
}
```

Figure 55 Source code of the *FinalizeAndAssemble* function

(2) Contract Asset Security

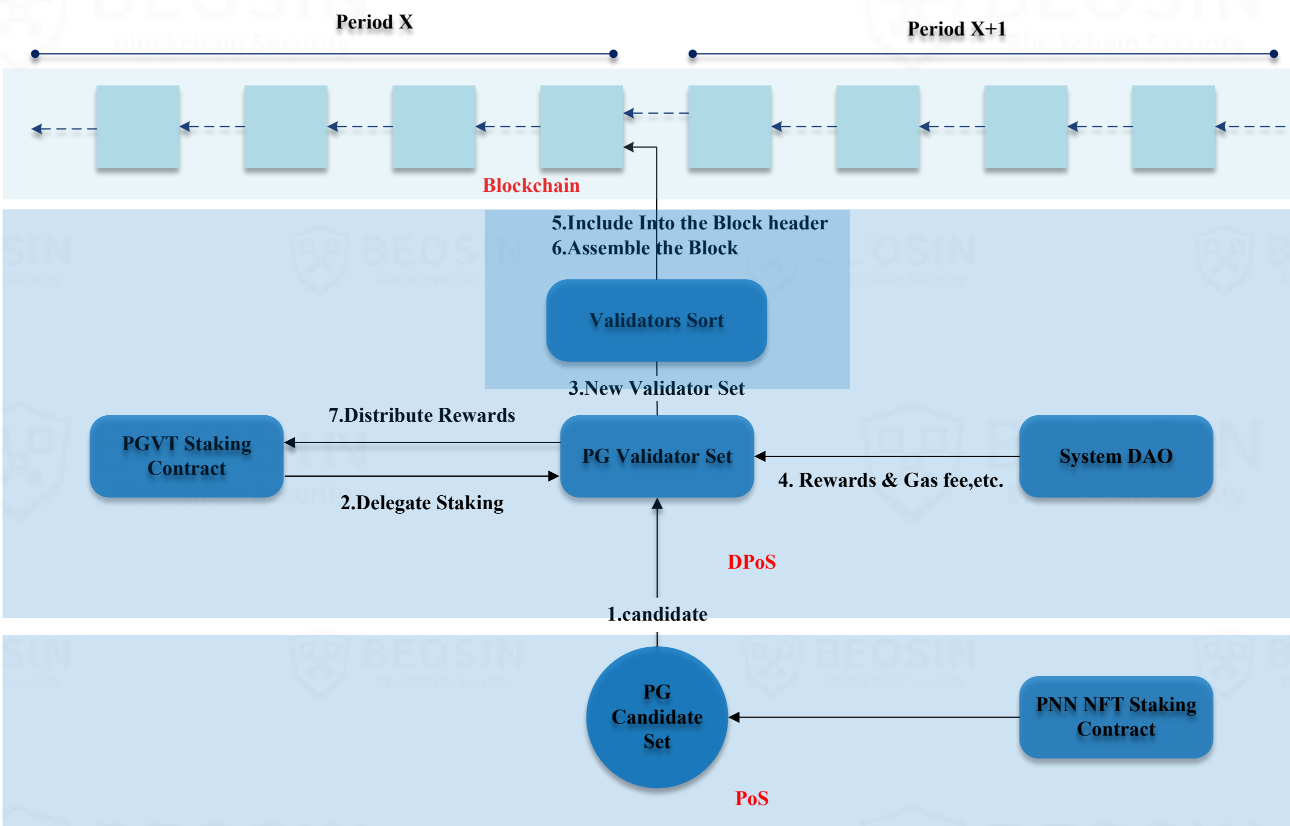
The PEGO supports EVM in order to be compatible with Ether, and the contracts on it are not upgradable or modifiable.

4 Consensus Security

PEGO's consensus mechanism PPOS is optimized on the basis of POS and DPOS. Its goal is to have fair treatment and rewards for all network participants (nodes and users). This mechanism incentivizes users to hold and stake PEGO tokens, which helps ensure that the network remains secure and sustainable over the long term. Consensus includes functions such as pledge, proposal voting, and election of verifiers.

Users can pledge the platform currency or the farm currency corresponding to the platform currency in the Pgvt contract to obtain Pgvt tokens in the contract, and then use pgvt tokens to pledge to a candidate node in the validator contract to obtain VT tokens. VT tokens can be Used to conduct governance votes on proposed proposals. After purchasing PNN NFT, the node can pay NFT to become a candidate node. The top 21 pledged candidates of the candidate node will become validators and have the authority to produce blocks. After the block is successfully produced, the validator will call the deposit function to distribute block dividends to all pledgers (not just validator supporters).

The detailed consensus architecture is shown in the figure below:



4.1 PoS Consensus Process Analysis

The PoS consensus part mainly deals with the logic of the candidate node. If an account needs to be elected as a candidate, it needs to convert a certain amount of ERC20 standard PNN tokens into ERC721 standard PNN NFT, and use the PNN NFT for the candidate node account staking tickets.

4.2 DPoS Consensus Process Analysis

The DPoS algorithm is divided into two parts: electing a group of block validators and scheduling production. The election process makes sure that stakeholders are ultimately in control because stakeholders lose the most when the network does not operate smoothly. Users need to pledge W3FT tokens and PG native tokens to obtain PGVT, and PGVT can be pledged to candidate nodes through agents. When the total amount of pledges ranks the top candidate nodes will become block validator nodes.

In the block generation phase, the selected nodes form a verification node set, and after sorting by size, each node will poll for block generation. And when the node produces a block, it will obtain the current block reward by calling the SystemDAO contract, and will send a part of the block reward to the mortgage contract, so that all accounts participating in the voting can get dividends.

- **Normal Operation**

Under normal operation block producers take turns producing a block every 3 seconds. Assuming no one misses their turn then this will produce the longest possible chain. It is invalid for a block producer to produce a block at any other time slot than the one they are scheduled for.

4.3 Logic Implementation of PoS

As shown below, to become a candidate node needs to stake minStaking amount of ERC20 standard PNN tokens through the *staking* function of the PNN contract to obtain ERC721 standard PNN NFT. Then staking the PNN NFT again through the *registerToValidator* function to become a candidate.

```
    function staking() external returns(uint256 tokenId) {
        require(tokenPnn != address(0),"tokenPnn not config");
        bool f = IPEP20(tokenPnn).transferFrom(msg.sender,address(this),minStaking);
        require(f,"token pnn not enough");
        tokenId = tokenIndex;
        _mint(msg.sender,tokenId);
        tokenIndex++;
    }
```

Figure 56 *staking* function

```
/**
 * Become the candidate of validator.
 */
function registerToValidator(uint256 tokenId) override external  returns(bool){

  IERC721(NFT_PNN_CONTRACT_ADDR).transferFrom(msg.sender,address(this),tokenId);

  address validator = msg.sender;
  PoolInfo storage vpool = pool[validator];
  require(stacking[validator] == 0,"already register to validator");
  vpool.validator = validator;
  vpool.v = 0;
  vpool.accPgPerShare = 0;
  vpool.enable = true;
  pool[validator] = vpool;
  stacking[validator] = tokenId;
  pendingWithdrawTime[validator] = block.timestamp.add(validatorLockTimeInMinutes * 1 minutes);
  emit registerToValidatorSuccess(validator,tokenId);
  return true;
}

```

Figure 57 *registerToValidator* function

4.4 Logic Implementation of DPoS

In the election phase, users can vote for candidates by calling the delegate function, and the candidates are sorted when voting. In the production phase, the node calls the getValidators function to obtain and fill it into the block header as a valid validator set, and polls to produce blocks.

```
/***************************** External functions *****************************/
function delegate(address validator, uint256 amount) override external noReentrant initParams {
  require(amount > 0, "invalid delegate amount");

  bool suc = IPEP20(PGVT_CONTRACT_ADDR).transferFrom(msg.sender,address(this),amount);
  require(suc, "transferFrom call fail");

  address delegator = msg.sender;

  uint256 value = candidateMap.get(validator);
  if (value>0) {
    candidateMap.put(validator,value.add(amount));
  } else {
    candidateMap.put(validator,amount);
  }

  esurePool(validator);
  delegatedOfValidator[delegator][validator] = delegatedOfValidator[delegator][validator].add(amount);
  pendingUndelegateTime[delegator][validator] = block.timestamp.add(vtLockTimeInMinutes * 1 minutes);

  if (accPgPerShare > 0 && delegated[delegator] > 0) {
    //将之前奖励打给用户
    uint256 pending = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION).sub(delegatorDebt[delegator]);
    delegated[delegator] = delegated[delegator].add(amount);
    delegatorDebt[delegator] = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION);//更新个人债务
    if (pending > 0) {
      (bool success,) = msg.sender.call{value: pending}("");
      require(success, "transfer reward failed");
    }
  } else {
    delegated[delegator] = delegated[delegator].add(amount);
    delegatorDebt[delegator] = delegated[delegator].mul(accPgPerShare).div(ACC_IPS_PRECISION);//更新个人债务
  }

  _mint(delegator,amount);
  emit delegateSuccess(delegator, validator, amount);

}
```

Figure 58 delegate function

```solidity
function getValidators() override external view returns(address[] memory) {
    uint256 m = candidateMap.listSize;
    uint valid = 0;

    address currentAddress = candidateMap.nextKey[address(1)];
    for(uint256 i = 0; i < m; ++i) {
        if (valid >= validatorNumber) {
            break;
        }
        if (validatorEnable(currentAddress)) {
            valid++;
        }
        currentAddress = candidateMap.nextKey[currentAddress];
    }

    address[] memory consensusAddrs = new address[](valid);
    uint256 j = 0;
    currentAddress = candidateMap.nextKey[address(1)];
    for(uint256 i = 0; i < m; ++i) {
        if (j >= validatorNumber) {
            break;
        }
        if (validatorEnable(currentAddress)) {
            consensusAddrs[j++] = currentAddress;
        }

        currentAddress = candidateMap.nextKey[currentAddress];
    }

    return consensusAddrs;
}
```

Figure 59 getValidators function

```go
func (p *Parlia) Prepare(chain consensus.ChainHeaderReader, header *types.Header) error {
    header.Coinbase = p.val
    header.Nonce = types.BlockNonce{}

    number := header.Number.Uint64()
    snap, err := p.snapshot(chain, number-1, header.ParentHash, nil)
    if err != nil {
        return err
    }

    // Set the correct difficulty
    header.Difficulty = CalcDifficulty(snap, p.val)

    // Ensure the extra data has all it's components
    if len(header.Extra) < extraVanity-nextForkHashSize {
        header.Extra = append(header.Extra, bytes.Repeat([]byte{0x00}, extraVanity-nextForkHashSize-len(header.Extra))...)
    }
    header.Extra = header.Extra[:extraVanity-nextForkHashSize]
    nextForkHash := forkid.NextForkHash(p.chainConfig, p.genesisHash, number)
    header.Extra = append(header.Extra, nextForkHash[:]...)

    if p.chainConfig.IsUnderCommunitySwap(header.Number) && number%p.config.Epoch == 0 {
        newValidators, err := p.getCurrentValidators(header.ParentHash, new(big.Int).Sub(header.Number, common.Big1))
        if err != nil {
            return err
        }
        // sort validator by address
        sort.Sort(validatorsAscending(newValidators))
        for _, validator := range newValidators {
            header.Extra = append(header.Extra, validator.Bytes()...)
        }
    }

    // add extra seal space
    header.Extra = append(header.Extra, make([]byte, extraSeal)...)
```

Figure 60 Parlia::Prepare() method in consensus engine

4.4 Rewards for Building Blocks

The amount of block rewards in PEGO chain is provided by the System DAO contract. The rewards are divided into two parts: one is the block rewards for validator nodes, and the other is the delegate staking rewards. The validator node reward will be sent directly to the validator when the block is assembled, and the delegate staking reward will be transferred through the validator node calling the *deposit* function in the consensus staking contract.

```go
func (p *Parlia) distributeIncoming(val common.Address, state *state.StateDB, header *types.Header, chain core.ChainContext,
    txs *[]*types.Transaction, receipts *[]*types.Receipt, receivedTxs *[]*types.Transaction, usedGas *uint64, mining bool) error {
    coinbase := header.Coinbase
    // issuing PG
    receiver, issuingAmount, err := p.getMintPg(header.ParentHash, new(big.Int).Sub(header.Number, common.Big1))
    if err != nil {
        return err
    }
    if receiver != nil && issuingAmount.Cmp(common.Big0) > 0 {
        state.AddBalance(*receiver, issuingAmount)
    }

    //区块奖励
    // block rewards
    blockRewards, err := p.getBlockRewards(header.ParentHash, new(big.Int).Sub(header.Number, common.Big1))
    if err != nil {
        return err
    }
    // getDelegatorRewardPercent
    delegatorRewardPercent0, err := p.getDelegatorRewardPercent(header.ParentHash, new(big.Int).Sub(header.Number, common.Big1))
    if err != nil {
        delegatorRewardPercent0 = big.NewInt(delegatorRewardPercent)
    }

    state.AddBalance(coinbase, blockRewards)
    totalReward := blockRewards
    //totalReward.Add(blockRewards, new(big.Int).SetUint64(*usedGas))
    if totalReward.Cmp(common.Big0) <= 0 {
        return nil
    }
    var delegatorReward = new(big.Int).Div(new(big.Int).Mul(totalReward, delegatorRewardPercent0), big.NewInt(10000))
    log.Trace("distribute to validator contract", "block hash", header.Hash(), "amount", delegatorReward)
    if delegatorReward.Cmp(common.Big0) > 0 {
        return p.distributeToValidator(delegatorReward, val, state, header, chain, txs, receipts, receivedTxs, usedGas, mining)
    }
    return nil
}
```

Figure 61 Calculate miner rewards

```go
func (p *Parlia) getBlockRewards(blockHash common.Hash, blockNumber *big.Int) (*big.Int, error) {
    // block
    blockNr := rpc.BlockNumberOrHashWithHash(blockHash, false)

    // method
    method := "getBlockRewards"

    ctx, cancel := context.WithCancel(context.Background())
    defer cancel() // cancel when we are finished consuming integers

    data, err := p.systemDaoABI.Pack(method)
    if err != nil {
        log.Error("Unable to pack tx for getBlockRewards", "error", err)
        return big.NewInt(0), err
    }
    // call
    msgData := (hexutil.Bytes)(data)
    toAddress := common.HexToAddress(systemcontracts.SystemDaoContract)
    gas := (hexutil.Uint64)(uint64(math.MaxUint64 / 2))
    result, err := p.ethAPI.Call(ctx, ethapi.TransactionArgs{
        Gas:  &gas,
        To:   &toAddress,
        Data: &msgData,
    }, blockNr, nil)
    if err != nil {
        return nil, err
    }

    var ret0 *big.Int
    if err := p.systemDaoABI.UnpackIntoInterface(&ret0, method, result); err != nil {
        return nil, err
    }
    return ret0, nil
}
```

Figure 62 *Parlia::getBlockRewards* method

```go
func (p *Parlia) distributeToValidator(amount *big.Int, validator common.Address,
    state *state.StateDB, header *types.Header, chain core.ChainContext,
    txs *[]*types.Transaction, receipts *[]*types.Receipt, receivedTxs *[]*types.Transaction, usedGas *uint64, mining bool) error {
    // method
    method := "deposit"

    // get packed data
    data, err := p.validatorSetABI.Pack(method)
    if err != nil {
        log.Error("Unable to pack tx for deposit", "error", err)
        return err
    }
    // get system message
    msg := p.getSystemMessage(header.Coinbase, common.HexToAddress(systemcontracts.ValidatorContract), data, amount)
    // apply message
    return p.applyTransaction(msg, state, header, chain, txs, receipts, receivedTxs, usedGas, mining)
}
```

Figure 63 *Parlia::distributeToValidator* method

5 Transaction Model Security

5.1 Transaction Processing Flow

PEGO's transaction processing is divided into two main parts: adding the transaction to the pool and executing the transaction. Adding a transaction to the pool means that each node adds the submitted transaction to the pool. As shown in the figure below, PEGO will perform the following checks on the transaction.

- Data size must be < 128KB
- Transaction amount must be non-negative (>=0)
- Ensure the transaction doesn't exceed the current block limit gas
- The signature data must be valid and the sender address can be resolved
- Gas price required to be transaction is not greater than the Gas price of the pool
- The nonce value of the transaction must be higher than the nonce value of the account on the current chain (lower than that means the transaction has been packaged)
- Current account balance must be greater than "Transaction Amount"
- Trading GAS requirements cannot be less than the specified quantity

```go
// validateTx checks whether a transaction is valid according to the consensus
// rules and adheres to some heuristic limits of the local node (price and size).
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Accept only legacy transactions until EIP-2718/2930 activates.
    if !pool.eip2718 && tx.Type() != types.LegacyTxType {
        return ErrTxTypeNotSupported
    }
    // Reject dynamic fee transactions until EIP-1559 activates.
    if !pool.eip1559 && tx.Type() == types.DynamicFeeTxType {
        return ErrTxTypeNotSupported
    }
    // Reject transactions over defined size to prevent DOS attacks
    if uint64(tx.Size()) > txMaxSize {
        return ErrOversizedData
    }
    // Transactions can't be negative. This may never happen using RLP decoded
    // transactions but may occur if you create a transaction using the RPC.
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }
    // Ensure the transaction doesn't exceed the current block limit gas.
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }
    // Sanity check for extremely large numbers
    if tx.GasFeeCap().BitLen() > 256 {
        return ErrFeeCapVeryHigh
    }
    if tx.GasTipCap().BitLen() > 256 {
        return ErrTipVeryHigh
    }
    // Ensure gasFeeCap is greater than or equal to gasTipCap.
    if tx.GasFeeCapIntCmp(tx.GasTipCap()) < 0 {
        return ErrTipAboveFeeCap
    }
    // Make sure the transaction is signed properly.
    from, err := types.Sender(pool.signer, tx)
    if err != nil {
        return ErrInvalidSender
    }
    // Drop non-local transactions under our own minimal accepted gas price or tip.
    pendingBaseFee := pool.priced.urgent.baseFee
    if !local && tx.EffectiveGasTipIntCmp(pool.gasPrice, pendingBaseFee) < 0 {
        return ErrUnderpriced
    }
    // Ensure the transaction adheres to nonce ordering
    if pool.currentState.GetNonce(from) > tx.Nonce() {
        return ErrNonceTooLow
    }
    // Transactor should have enough funds to cover the costs
    // cost == V + GP * GL
    if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
        return ErrInsufficientFunds
    }
    // Ensure the transaction has more gas than the basic tx fee.
    intrGas, err := IntrinsicGas(tx.Data(), tx.AccessList(), tx.To() == nil, true, pool.istanbul)
    if err != nil {
        return err
    }
    if tx.Gas() < intrGas {
        return ErrIntrinsicGas
    }

    // Contract creation transaction
    if tx.To() == nil && pool.chainconfig.Consortium != nil {
        whitelisted := state.IsWhitelistedDeployer(pool.currentState, from)
        if !whitelisted {
            return ErrUnauthorizedDeployer
        }
    }

    // Check if sender and recipient are blacklisted
    if pool.chainconfig.Consortium != nil && pool.odysseus {
        contractAddr := pool.chainconfig.BlacklistContractAddress
        if state.IsAddressBlacklisted(pool.currentState, contractAddr, &from) || state.IsAddressBlacklisted(pool.currentState, contractAddr, tx.To()) {
            return ErrAddressBlacklisted
        }
    }

    return nil
}
```

Figure 64 Partial source code of transaction check

5.2 Transaction replay audit

Replay attack refers to when the blockchain is hard-forked, the blockchain has a permanent difference and produces two historical transactions, on the exact corresponding chains, with differences such as address, private key, and balance, in which the same transaction can take effect at the same time on the two chains.

After testing, the trades on the PEGO chain (including EVM transactions) cannot be replayed, and during the transaction when adding to transaction pool, validity is verified, while illegal transactions are rejected because the transaction data format does not match; Chains with substrate frameworks can also fail transactions because of differences in chain ids. Therefore, it is recommended that the project developer pay close attention to chain id to avoid duplicate ids for the same type of chain.

5.3 Dusting Attack

The dusting attack is when an attacker sends a very small amount of tokens, called "dust", to a user's wallet. By tracking the dusted wallet funds and all transactions, the attacker can then connect to these addresses and determine the company or person to whom these wallet addresses belong, destroying the anonymity of the block chain; or misuse the block chain resources, causing the block chain memory pool strain. If the dust money is not moved, the attacker cannot establish a connection to it and cannot complete the de-anonymization of the wallet or address owners.

The PEGO chain is based on the account model, so dust attacks are not considered.

5.4 Trading Flooding Attacks

In PEGO, transactions are subject to a fee. The base fee for creating a contract is 53,000, and the base fee for a normal transaction is 21,000.

```
const (
    GasLimitBoundDivisor uint64 = 256                // The bound divisor of the gas limit, used in update calculations.
    MinGasLimit          uint64 = 5000               // Minimum the gas limit may ever be.
    MaxGasLimit          uint64 = 0x7fffffffffffffff // Maximum the gas limit (2^63-1).
    GenesisGasLimit      uint64 = 4712388            // Gas limit of the Genesis block.

    MaximumExtraDataSize  uint64 = 32     // Maximum size extra data may be after Genesis.
    ForkIDSize            uint64 = 4      // The length of fork id
    ExpByteGas            uint64 = 10     // Times ceil(log256(exponent)) for the EXP instruction.
    SloadGas              uint64 = 50     // Multiplied by the number of 32-byte words that are copied (round up) for any *COPY operation and added.
    CallValueTransferGas  uint64 = 9000   // Paid for CALL when the value transfer is non-zero.
    CallNewAccountGas     uint64 = 25000  // Paid for CALL when the destination address didn't exist prior.
    TxGas                 uint64 = 21000  // Per transaction not creating a contract. NOTE: Not payable on data of calls between transactions.
    SystemTxsGas          uint64 = 500000 // The gas reserved for system txs; only for parlia consensus
    TxGasContractCreation uint64 = 53000  // Per transaction that creates a contract. NOTE: Not payable on data of calls between transactions.
    TxDataZeroGas         uint64 = 4      // Per byte of data attached to a transaction that equals zero. NOTE: Not payable on data of calls between transactions.
    QuadCoeffDiv          uint64 = 512    // Divisor for the quadratic particle of the memory cost equation.
    LogDataGas            uint64 = 8      // Per byte in a LOG* operation's data.
    CallStipend           uint64 = 2300   // Free gas given at beginning of call.
```

Figure 65 Transactions minimum gas consumption

However, it is worth mentioning that gasprice is set by the System DAO contract and if it is 0, it will lead to trading flooding attacks.

```
function setBaseGasPrice(uint256 price) override external onlyGov returns(bool) {
    emit gasPriceChanged(gasPrice, price);
    gasPrice = price;
    return true;
}
```

Figure 66 *setBaseGasPrice* function in System DAO contract

5.5 Double-spending Attack

For PEGO's transactions, each transaction of the account contains a unique and mintable nonce, and PEGO uses the consensus model of PoSA, it is also difficult to complete a double-spending through a 51% attack.

5.6 Illegal Transactions

The user will sign the entire transaction data when initiating a transaction, and any changes to the data in the transaction will cause the PEGO node to fail the signature check.

```go
func (c *Consortium) verifySeal(chain consensus.ChainHeaderReader, header *types.Header, parents []*types.Header, snap *Snapshot) error {
    // Verifying the genesis block is not supported
    number := header.Number.Uint64()
    if number == 0 {
        return consortiumCommon.ErrUnknownBlock
    }

    // Resolve the authorization key and check against validators
    signer, err := ecrecover(header, c.signatures, c.chainConfig.ChainID)
    if err != nil {
        return err
    }

    if signer != header.Coinbase {
        return errCoinBaseMisMatch
    }

    if _, ok := snap.Validators[signer]; !ok {
        return errUnauthorizedValidator
    }

    for seen, recent := range snap.Recents {
        if recent == signer {
            // Signer is among recents, only fail if the current block doesn't shift it out
            if limit := uint64(len(snap.Validators)/2 + 1); seen > number-limit {
                return consortiumCommon.ErrRecentlySigned
            }
        }
    }

    // Ensure that the difficulty corresponds to the turn-ness of the signer
    if !c.fakeDiff {
        inturn := snap.inturn(signer)
        if inturn && header.Difficulty.Cmp(diffInTurn) != 0 {
            return consortiumCommon.ErrWrongDifficulty
        }
        if !inturn && header.Difficulty.Cmp(diffNoTurn) != 0 {
            return consortiumCommon.ErrWrongDifficulty
        }
    }

    return nil
}
```

Figure 67 Transaction signature check(1/2)

```go
// ecrecover extracts the Ronin account address from a signed header.
func ecrecover(header *types.Header, sigcache *lru.ARCCache, chainId *big.Int) (common.Address, error) {
    // If the signature's already cached, return that
    hash := header.Hash()
    if address, known := sigcache.Get(hash); known {
        return address.(common.Address), nil
    }
    // Retrieve the signature from the header extra-data
    if len(header.Extra) < consortiumCommon.ExtraSeal {
        return common.Address{}, consortiumCommon.ErrMissingSignature
    }
    signature := header.Extra[len(header.Extra)-consortiumCommon.ExtraSeal:]

    // Recover the public key and the Ethereum address
    pubkey, err := crypto.Ecrecover(SealHash(header, chainId).Bytes(), signature)
    if err != nil {
        return common.Address{}, err
    }
    var signer common.Address
    copy(signer[:], crypto.Keccak256(pubkey[1:])[12:])

    sigcache.Add(hash, signer)
    return signer, nil
}
```

Figure 68 Transaction signature check(2/2)

If the PEGO node is malicious, the signature checks fails here, but the verification node also checks the transaction signature again, and the forged transaction will fail to be verified.

5.7 Fake Deposit Attack

Fake recharge attack is the transaction execution failure, but the corresponding receipt status shows success. In PEGO, there are only two types of transaction status, Failed(0x0) and Success(0x1). For transactions that pass the check, both gas exceeds the block limit and transaction execution fails will return failed, and there is no chain-level fake deposit attack.


```
{
  blockHash: "0x4882f8dea0cc9f4828ce3457b114f9ee33b82dfa36d9869d285eabe43bcd68d2",
  blockNumber: 1187,
  contractAddress: null,
  cumulativeGasUsed: 21000,
  effectiveGasPrice: 1000000000,
  from: "0x8c21b2c895ce15ed5099fcbe465f688e6f282dcb",
  gasUsed: 21000,
  logs: [],
  logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  status: "0x0",
  to: "0xa131ad247055fd2e2aa8b156a11bdec81b9ead95",
  transactionHash: "0xb362cd9a06525212c53aad178fe4c4ad88b54a06bda93ac00364739bc978f1b1",
  transactionIndex: 0,
  type: "0x0"
```


Figure 69 Transaction receipt information(1/2)


```
{
  blockHash: "0xb45c6cab777347853d141d0043ac64bb2b2411dd3725dde9e518125ce26fb4ec",
  blockNumber: 1141,
  contractAddress: null,
  cumulativeGasUsed: 21000,
  effectiveGasPrice: 1000000000,
  from: "0x8c21b2c895ce15ed5099fcbe465f688e6f282dcb",
  gasUsed: 21000,
  logs: [],
  logsBloom: "0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  status: "0x1",
  to: "0x9fb29aac15b9a4b7f17c3385939b007540f4d791",
  transactionHash: "0x5cf8593b5d59165d9e01f36180c6c00e0af2d382bf26fd6091a6484a9bf74ef9",
  transactionIndex: 0,
  type: "0x0"
```


Figure 70 Transaction receipt information(2/2)

5.8 Contract trading security

PEGO is compatible with EVM transactions. Our team tested various types of EVM contract transactions on the PEGO chain and found that Ethereum contract transactions have no security issues and can run on the PEGO chain.

Historical Vulnerability Detection

The PEGO chain has completed the repair of the transaction pool DOS vulnerability. However, it is still recommended that the project party pay attention to the third-party components/libraries that the PEGO chain depends on, so as to prevent unsafe third-party components/libraries from causing harm to the PEGO network.

Consensus Contracts Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

Appendix

1.1 Vulnerability Assessment Metrics and Status in Smart Contracts

1.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Impact / Likelihood	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

1.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

1.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

1.2 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

1.3 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

Official Website
https://www.beosin.com
Telegram
https://t.me/+dD8Bnqd133RmNWNl
Twitter
https://twitter.com/Beosin_com
Email
Contact@beosin.com