成都链安
BEOSIN

Smart contract security audit report

Audit Number：202010231752

Report Query Name: SNP

Smart Contract Name And Address Link：

SnpToken.sol: https://github.com/Seele-N/NervLedger-Contract/blob/main/contracts/SnpToken.sol

SnpMaster.sol: https://github.com/Seele-N/NervLedger-Contract/blob/main/contracts/SnpMaster.sol

Migrations.sol: https://github.com/Seele-N/NervLedger-Contract/blob/main/contracts/Migrations.sol

Commit Hash： 8d6ee0277959250c02fd7ded3285a0ff74177e0d

Start Date：2020.10.20

Completion Date：2020.10.23

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass
		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass

		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

Note: Audit results and suggestions in code comments

Disclaimer: This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin (Chengdu LianAn) Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin (Chengdu LianAn) Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin (Chengdu LianAn) Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts SnpToken, SnpMaster, and Migrations, including Coding Standards, Security, and Business Logic. **The SnpToken, SnpMaster and Migrations contracts pass all audit items. The overall result is Pass.** The smart contract is able to function properly.

1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- Result: Pass

1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing ETH.

- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.
- Result: Pass

2.10 Replay Attack

- Description: Check the weather the implement possibility of Replay Attack exists in the contract.
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Result: Pass

3. Business Security

Check whether the business is secure.

3.1 Business analysis of Contract SnpToken

(1) Basic Token Information

Token name	SNP Token
Token symbol	SNP
decimals	18
totalSupply	Initial supply is 0 (Mintable , burnable, the maximum token total supply is 3000000)
Token type	ERC20

Table 1 Basic Token Information of SNP

(2) ERC20 Token Standard Functions

- Description: The SnpToken Contracts implement a Token which conforms to the ERC20 Standards. It should be noted that the user can directly call the approve function to set the approval value for the specified address, but in order to avoid multiple authorizations, it is recommended to use the increaseAllowance and decreaseAllowance functions when modifying the approval value, instead of using the approve function directly.
- Related functions: *name, symbol, decimals, totalSupply, balanceOf, allowance, transfer, transferFrom, approve, increaseAllowance, decreaseAllowance. burn*
- Result: Pass

(3) mint function and mint authority management

- Description: As shown in Figure 1 below, the user or contract with mint permission can call *mint* function to mint tokens to the specified address. The maximum token total supply of SNP is 3000000. The contract owner can set the minter of SNP.

```
61  function mint(address account, uint256 amount)
62      public
63      virtual
64      override
65      whenNotPaused
66      returns (uint256)
67  {
68      require(minters[msg.sender], "SnpToken: You are not the minter");
69      uint256 supply = _totalSupply.add(amount);
70      if (supply > TOTAL_SUPPLY) {
71          supply = TOTAL_SUPPLY;
72      }
73      amount = supply.sub(_totalSupply);
74      _mint(account, amount);
75      return amount;
76  }
```

Figure 1 mint Function Source Code

- Related functions: *mint, updatePool, balanceOf, getTotalReward*
- Result: Pass

(4) withdraw function of Contract SnpToken

- Description: As shown in Figure 2 below, the contract implements the withdraw function to claim token. Owner can withdraw any tokens that transferred to this contract address.

```
171  function withdraw(address token, uint256 amount) public onlyOwner {
172      IERC20(token).safeTransfer(msg.sender, amount);
173  }
```

Figure 2 withdraw Function Source Code

- Related functions: *withdraw*
- Result: Pass

3.2 Business analysis of Contract Migrations

(1) setCompleted Function

- Description: As shown in Figure 3 below, the contract implements the *setCompleted* function for contract owner to set the value of variable last_completed_migration.

```
16  function setCompleted(uint completed) public restricted {
17      last_completed_migration = completed;
18  }
19  }
```

Figure 3 setCompleted Function Source Code

- Related functions: *setCompleted*
- Result: Pass

3.3 Business analysis of Contract SnpMaster

(1) add Function

- Description: As shown in Figure 4 below, the contract implements the *add* function to add the Pool. The contract owner can call this function to add the Pool for the user to stake for getting the reward and store the pool-related information.

```
107   // Add a new lp to the pool. Can only be called by the owner.
108   // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
109   function add(
110       uint256 _allocPoint,
111       IERC20 _lpToken,
112       bool _withUpdate
113   ) public onlyOwner {
114       if (_withUpdate) {
115           massUpdatePools();
116       }
117       uint256 lastRewardBlock = block.number > startBlock
118           ? block.number
119           : startBlock;
120       totalAllocPoint = totalAllocPoint.add(_allocPoint);
121       poolInfo.push(
122           PoolInfo({
123               lpToken: _lpToken,
124               allocPoint: _allocPoint,
125               lastRewardBlock: lastRewardBlock,
126               lpSupply: 0,
127               accSnpPerShare: 0,
128               lockPeriod: 0,
129               unlockPeriod: 0,
130               emergencyEnable: false
131           })
132       );
133   }
```

Figure 4 add Function Source Code

- Related functions: *add, massUpdatePools*
- Result: Pass

(2) set Function

- Description: As shown in Figure 5 below, contract implements *set* function to set the reward allocation point of the specified pool, the contract owner can call this function to set the reward allocation point of the specified pool. After the pool reward allocation point is modified, it will affect the value of SNP rewards when users withdraw or deposit tokens.

```
159  function set(
160      uint256 _pid,
161      uint256 _allocPoint,
162      bool _withUpdate
163  ) public onlyOwner {
164      if (_withUpdate) {
165          massUpdatePools();
166      }
167      totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
168          _allocPoint
169      );
170
171      PoolInfo storage pool = poolInfo[_pid];
172      if (pool.lpSupply > 0) {
173          uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
174          uint256 lpSupply = pool
175              .lpSupply
176              .mul(pool.allocPoint)
177              .mul(1e18)
178              .div(100)
179              .div(10**lpDec);
180          totallpSupply = totallpSupply.sub(lpSupply);
181
182          lpSupply = pool.lpSupply.mul(_allocPoint).mul(1e18).div(100).div(
183              10**lpDec
184          );
185          totallpSupply = totallpSupply.add(lpSupply);
186      }
187
188      poolInfo[_pid].allocPoint = _allocPoint;
189  }
```

Figure 5 set Function Source Code

- Related functions: *set*
- Result: Pass

(3) getMultiplier Function

- Description: As shown in Figure 6 below, contract implements the *getMultiplier* function to return reward multiplier the given _from to _to block. If the value of the variable _to is greater than the endBlock, return the reward multiplier over the given _from to endBlock. In addition, when _from is greater than endBlock, the return value is 0.

```
236  function getMultiplier(uint256 _from, uint256 _to)
237      public
238      view
239      returns (uint256)
240  {
241      uint256 toFinal = _to > endBlock ? endBlock : _to;
242      if (_from >= endBlock) {
243          return 0;
244      }
245      return toFinal.sub(_from);
246  }
```

Figure 6 getMultiplier Function Source code

- Related functions: *getMultiplier*
- Result: Pass

(4) updatePool Function

- Description: As shown in Figure 7 below, contract implements *updatePool* function to update pool SNP rewards and information of current block. Any user can call this function to update latest pool SNP rewards and information, and call *mint* function to mint all SNP rewards generated after last block update to this contract address. 49% of the calculated amount of tokens to be minted will be sent to ecosystemrate address(the contract owner can call the *setSeeleEcosystem* function to set the value of address ecosystemrate), and 51% will be sent to the address of this contract. In addition, anyone can update all pools at once by calling the *massUpdatePools* function.

```
200  function updatePool(uint256 _pid) public {
201      PoolInfo storage pool = poolInfo[_pid];
202      if (block.number <= pool.lastRewardBlock) {
203          return;
204      }
205      uint256 lpSupply = pool.lpSupply;
206      if (lpSupply == 0) {
207          pool.lastRewardBlock = block.number;
208          return;
209      }
210
211      uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
212      uint256 lpSupply1e18 = lpSupply.mul(1e18).div(10**lpDec);
213
214      uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
215      uint256 snpmint = multiplier
216          .mul(snpPerBlock)
217          .mul(pool.allocPoint)
218          .mul(lpSupply1e18)
219          .div(100)
220          .div(totallpSupply);
221
222      snptoken.mint(seeleEcosystem, snpmint.mul(ecosystemrate).div(100));
223
224      uint256 snpReward = snpmint.mul(farmrate).div(100);
225      snpReward = snptoken.mint(address(this), snpReward);
226
227      totalMintReward = totalMintReward.add(snpReward);
228
229      pool.accSnpPerShare = pool.accSnpPerShare.add(
230          snpReward.mul(1e12).div(lpSupply)
231      );
232      pool.lastRewardBlock = block.number;
233  }
```

Figure 7 updatePool Function Source Code

- Related functions: *updatePool, getMultiplier, mint*
- Result: Pass

(5) deposit Function

- Description: As shown in Figure 8 below, the contract implements the *deposit* function for users to stake tokens, the user pre-approves this contract address and then calls this function to deposit tokens(require the pool is exist). Update the pool information when the user is deposited, if the user has previous deposit, calculate the user's previous deposit reward and send the reward to the user address. If

the user deposit multiple times within a day to enter the pool without lock-up period, additional fees will be charged and sent to the governance address(the contract owner can call the *setGovernance* function to set the value of address governance).

```
282  function deposit(
283      uint256 _pid,
284      uint256 _amount,
285      string calldata _refuser
286  ) public whenNotPaused {
287      PoolInfo storage pool = poolInfo[_pid];
288      UserInfo storage user = userInfo[_pid][msg.sender];
289      updatePool(_pid);
290      if (user.amount > 0) {
291          uint256 pending = user
292              .amount
293              .mul(pool.accSnpPerShare)
294              .div(1e12)
295              .sub(user.rewardDebt);
296          if (pending > 0 && pool.lockPeriod == 0) {
297              uint256 _depositTime = now - user.depositTime;
298              if (_depositTime < 1 days) {
299                  uint256 _actualReward = _depositTime
300                      .mul(pending)
301                      .mul(1e18)
302                      .div(1 days)
303                      .div(1e18);
304                  uint256 _goverAomunt = pending.sub(_actualReward);
305                  safeSnpTransfer(governance, _goverAomunt);
306                  pending = _actualReward;
307              }
308              safeSnpTransfer(msg.sender, pending);
309          }
310      }
311      if (_amount > 0) {
312          pool.lpToken.safeTransferFrom(
313              address(msg.sender),
314              address(this),
315              _amount
316          );
317          user.amount = user.amount.add(_amount);
318          pool.lpSupply = pool.lpSupply.add(_amount);
319          user.depositTime = now;
320          user.refAddress = _refuser;
321          uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
322          uint256 lpSupply = _amount
323              .mul(pool.allocPoint)
324              .mul(1e18)
325              .div(100)
326              .div(10**lpDec);
327          totallpSupply = totallpSupply.add(lpSupply);
328      }
329      user.rewardDebt = user.amount.mul(pool.accSnpPerShare).div(1e12);
330      emit Deposit(msg.sender, _pid, _amount, user.refAddress);
331  }
```

Figure 8 deposit Function Source Code

- Related functions: *deposit, updatePool, safeSnpTransfer*
- Result: Pass

(6) withdraw Function

- Description: As shown in Figure 9 below, the contract implements the *withdraw* function for users to withdraw deposit tokens and SNP rewards, the user can call this function to withdraw the specified amount of deposit tokens and all SNP reward in the current block. Update pool information when users withdraw deposit tokens and SNP rewards, and transfer the specified deposited tokens and SNP rewards to the user address and update the user deposit information. If the user withdraws within one day after the deposit, additional fees will be charged and sent to the governance address(the contract owner can call the setGovernance function to set the value of address governance).

```
334  function withdraw(uint256 _pid, uint256 _amount) public {
335      PoolInfo storage pool = poolInfo[_pid];
336      UserInfo storage user = userInfo[_pid][msg.sender];
337      require(user.amount >= _amount, "withdraw: not good amount");
338      if (_amount > 0 && pool.lockPeriod > 0) {
339          require(
340              now >= user.depositTime + pool.lockPeriod,
341              "withdraw: lock time not reach"
342          );
343          if (pool.unlockPeriod > 0) {
344              require(
345                  (now - user.depositTime) % pool.lockPeriod <=
346                      pool.unlockPeriod,
347                  "withdraw: not in unlock time period"
348              );
349          }
350      }
351
352      updatePool(_pid);
353      uint256 pending = user.amount.mul(pool.accSnpPerShare).div(1e12).sub(
354          user.rewardDebt
355      );
356      if (pending > 0) {
357          uint256 _depositTime = now - user.depositTime;
358          if (_depositTime < 1 days && pool.lockPeriod == 0) {
359              uint256 _actualReward = _depositTime
360                  .mul(pending)
361                  .mul(1e18)
362                  .div(1 days)
363                  .div(1e18);
364              uint256 _goverAomunt = pending.sub(_actualReward);
365              safeSnpTransfer(governance, _goverAomunt);
366              pending = _actualReward;
367          }
368          safeSnpTransfer(msg.sender, pending);
369      }
370      if (_amount > 0) {
371          user.amount = user.amount.sub(_amount);
372          pool.lpSupply = pool.lpSupply.sub(_amount);
373          pool.lpToken.safeTransfer(address(msg.sender), _amount);
374
375          uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
376          uint256 lpSupply = _amount
377              .mul(pool.allocPoint)
378              .mul(1e18)
379              .div(100)
380              .div(10**lpDec);
381          totallpSupply = totallpSupply.sub(lpSupply);
382      }
383      user.rewardDebt = user.amount.mul(pool.accSnpPerShare).div(1e12);
384      emit Withdraw(msg.sender, _pid, _amount);
385  }
```

Figure 9 withdraw Function Source Code

- Related functions: *withdraw*, *safeSnpTransfer, safeTransfer*
- Result: Pass

(7) emergencyWithdraw Function

- Description: As shown in Figure 10 below, the contract implements the *emergencyWithdraw* function for users to withdraw deposited tokens. The user can call this function to withdraw all deposit tokens in the pool with no lock-up period or allow emergency withdrawal (the contract owner can call the *setPoolEmergencyEnable* function to set whether to allow emergency withdrawal). Update user deposit information and transfer all deposited tokens to the user address(Note: calling this function cannot get any deposit rewards).

```
388  function emergencyWithdraw(uint256 _pid) public {
389      PoolInfo storage pool = poolInfo[_pid];
390      UserInfo storage user = userInfo[_pid][msg.sender];
391      require(
392          pool.lockPeriod == 0 || pool.emergencyEnable == true,
393          "emergency withdraw: not good condition"
394      );
395      pool.lpToken.safeTransfer(address(msg.sender), user.amount);
396
397      uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
398      uint256 lpSupply = user
399          .amount
400          .mul(pool.allocPoint)
401          .mul(1e18)
402          .div(100)
403          .div(10**lpDec);
404      totallpSupply = totallpSupply.sub(lpSupply);
405
406      emit EmergencyWithdraw(msg.sender, _pid, user.amount);
407
408      user.amount = 0;
409      user.rewardDebt = 0;
410  }
```

Figure 10 emergencyWithdraw Function Source Code

- Related functions: *emergencyWithdraw, safeTransfer*
- Result: Pass

(8) pendingSnp function

- Description: As shown in Figure 11 below, the contract implements the *pendingSnp* function for users to query the number of SNP rewards that can be obtained.

```
249  function pendingSnp(uint256 _pid, address _user)
250      external
251      view
252      returns (uint256)
253  {
254      PoolInfo storage pool = poolInfo[_pid];
255      UserInfo storage user = userInfo[_pid][_user];
256      uint256 accSnpPerShare = pool.accSnpPerShare;
257      uint256 lpSupply = pool.lpSupply;
258      if (block.number > pool.lastRewardBlock && lpSupply != 0) {
259          uint256 lpDec = ILPERC20(address(pool.lpToken)).decimals();
260          uint256 lpSupply1e18 = lpSupply.mul(1e18).div(10**lpDec);
261
262          uint256 multiplier = getMultiplier(
263              pool.lastRewardBlock,
264              block.number
265          );
266          uint256 snpmint = multiplier
267              .mul(snpPerBlock)
268              .mul(pool.allocPoint)
269              .mul(lpSupply1e18)
270              .div(100)
271              .div(totallpSupply);
272
273          uint256 snpReward = snpmint.mul(farmrate).div(100);
274          accSnpPerShare = accSnpPerShare.add(
275              snpReward.mul(1e12).div(lpSupply)
276          );
277      }
278      return user.amount.mul(accSnpPerShare).div(1e12).sub(user.rewardDebt);
279  }
```

Figure 11 pendingSnp Function Source Code

- Related functions: *pendingSnp, getMultiplier*
- Result: Pass

(9) Parameter Related Function

- Description: As shown in Figures 12, 13, 14, and 15, the contract implements the functions of *setSnpPerBlock, setEndMintBlock, setPoolEmergencyEnable,* and *setPoolLockTime* function to set pool related parameters. The contract owner can call these functions to set the number of SNP tokens generated in each block, the mining end block, whether the pool allows emergency withdrawal, and the lock period and unlock time of every pool. The contract owner can modify the relevant parameters of the pool at any time, which will affect the user's reward.

```
423  function setSnpPerBlock(uint256 _snpPerBlock) public onlyOwner {
424      require(_snpPerBlock > 0, "!snpPerBlock-0");
425
426      snpPerBlock = _snpPerBlock;
427  }
428  }
```

Figure 12 setSnpPerBlock Function Source Code

```
154  function setEndMintBlock(uint256 _endBlock) public onlyOwner {
155      endBlock = _endBlock;
156  }
```

Figure 13 setEndMintBlock Function Source Code

```
146  function setPoolEmergencyEnable(uint256 _pid, bool _emergencyEnable)
147      public
148      onlyOwner
149  {
150      poolInfo[_pid].emergencyEnable = _emergencyEnable;
151  }
```

Figure 14 setPoolEmergencyEnable Function Source Code

```
136  function setPoolLockTime(
137      uint256 _pid,
138      uint256 _lockPeriod,
139      uint256 _unlockPeriod
140  ) public onlyOwner {
141      poolInfo[_pid].lockPeriod = _lockPeriod;
142      poolInfo[_pid].unlockPeriod = _unlockPeriod;
143  }
```

Figure 15 setPoolLockTime Function Source Code

- Related functions: *setSnpPerBlock, setEndMintBlock, setPoolEmergencyEnable, setPoolLockTime*
- Result: Pass

4. Conclusion

Beosin(Chengdu LianAn) conducted a detailed audit on the design and code implementation of the smart contracts Migrations, SnpMaster and SnpToken. All problems found during the audit have been notified to the project party, and the project party believes that no repair is needed. In the SnpToken contract, the owner can call the *addMinter* function to add minters, and the minter can mint a specified amount of tokens to the specified address. In the SnpMaster contract, the owner can call the *setSnpPerBlock* function to modify the

SNP token output of each block, which may affect the user's reward. After communicating with the project party, they said that this is their normal design, because they may accelerate the production of blocks in the later stage. In addition, the owner can also call the *set, setEndMintBlock, setPoolEmergencyEnable* and *setPoolLockTime* functions to modify the relevant parameters of each pool, which may also affect the user's reward. The overall audit result of the smart contracts Migrations, SnpMaster and SnpToken is **Pass**.

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com