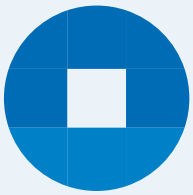
01 JUN 2018



Onther Inc.

SMART CONTRACT AUDIT REPORT 2

HAECHILABS

01. INTRODUCTION

본 보고서는 Onther 팀이 제작한 토큰과 PoS 시스템과 유사한 이자 토큰 생성 스마트 컨트랙트의 보안을 감사하기 위해 작성되었습니다. HAECHI Labs 팀에서는 Onther 팀이 제작한 스마트 컨트랙트의 구현 및 설계가 보안상 안전한지에 중점을 맞춰 감사를 진행했습니다.

Audit 에 사용된 코드는 “Onther-Tech/minime-pos-controller” Github 저장소(https://github.com/Onther-Tech/minime-pos-controller) 에서 찾아볼 수 있습니다. Audit 에 사용된 코드의 마지막 커밋은 “e65681e0213339928e21c79634376fa664544375” 입니다. 해당 코드중 “zeppelin”, “minime” 폴더에 있는 오픈소스 라이브러리 코드는 audit 의 대상이 아닙니다.

코드 저장소 링크: https://github.com/Onther-Tech/minime-pos-controller

02. AUDITED FILE

- interfaces/POSTokenI.sol
- misc/Migrations.sol
- BalanceUpdatableMiniMeToken.sol
- POSController.sol
- POSFactory.sol
- POSMiniMeToken.sol
- POSMintableToken.sol
- POSTokenAPI.sol

03. ABOUT “HAECHI LABS”

“HAECHI Labs” 는 기술을 통해 건강한 블록체인 생태계에 기여하자는 비전을 가지고 있습니다. HAECHI Labs는 스마트 컨트랙트의 보안 뿐만 아니라 블록체인 기술에 깊이있는 연구를 진행하고 있습니다. The DAO, Parity Multisig Wallet, SmartMesh(ERC20) 해킹 사건과 같이 스마트 컨트랙트의 보안 취약점을 이용한 사건들이 지속적으로 발생하고 있습니다. “HAECHI Labs”는 이러한 보안 사고 등을 예방하기 위해 안전한 스마트 컨트랙트 설계와 구현 및 보안 감사에 최선을 다합니다. 고객사가 목적에 맞는 안전한 스마트 컨트랙트를 구현하고 운영시 발생하는 가스비를 최적화할 수 있도록 스마트 컨트랙트 관련 서비스를 제공합니다. “HAECHI Labs” 는 스타트업에서 다년간 Software Engineer로 개발을 하고 서울대학교 블록체인 학회 Decipher 와 nonce research 에서 블록체인 연구를 한 사람들로 구성되어 있습니다.

04. ISSUES FOUND

HAECHI Labs는 Onther 팀이 발견된 모든 이슈에 대하여 개선 하는 것을 권장합니다. 이어지는 이슈 설명에서는 코드를 세부적으로 지칭하기 위해서 {파일 이름}:{줄 번호} 포맷을 사용할 것입니다. 예를 들면, Token.sol:20은 Token.sol 파일의 23번째 줄을 지칭합니다.

1. POSController.sol 의 생성자에서 *_posInterval* 에 0이 올 수 있습니다.

```
37    /* Constructor */
38    function POSController(
39      address _token,
40      uint256 _posInterval,
41      uint256 _initBlockNumber,
42      uint256 _posRate,
43      uint256 _posCoeff
44    ) public {
45      require(_token != address(0));
46
47      token = _token;
48      posInterval = _posInterval;
49      posRate = _posRate;
50      posCoeff = _posCoeff;
51
52      if (_initBlockNumber == 0) {
53        initBlockNumber = block.number;
54      } else {
55        initBlockNumber = _initBlockNumber;
56      }
57    }
```

POSController.sol

POSController.sol 에서 *posInterval* 은 0 보다 큰 값이 와야됩니다. POSController.sol:153, POSController.sol:158 을 보면 *getClaimRate* 함수 내에서 *posInterval* 값으로 나눗셈을 해서 몫, 나머지를 구하는데 0 을 분모에 사용하는것이기 때문에 예외가 발생합니다. 따라서 46번째 줄에 *requrie(_posInterval > 0)* 과 같은 조건문을 넣는것을 권장합니다.

2. POSController.sol 의 *doClaim* 함수에서 *calimedValue* 에 저장하는 값을 형 변환하는 것은 올바르지 않습니다.

```
100     /* Internal */
101     function doClaim(address _owner, Claim[] storage c) internal {
102       uint256 claimRate;
103
104       if (c.length == 0 && claimable(block.number)) {
105         claimRate = getClaimRate(0);
106       } else if (c.length > 0 && claimable(c[c.length - 1].fromBlock)) {
107         claimRate = getClaimRate(c[c.length - 1].fromBlock);
108       }
109
110       if (claimRate > 0) {
111         Claim storage newClaim = c[c.length++];
112
113         // TODO: reduce variables into few statements
114         uint256 balance = ERC20(token).balanceOf(_owner);
115
116         uint256 targetBalance = balance.mul(posCoeff.add(claimRate)).div(posCoeff);
117         uint256 claimedValue = targetBalance.sub(balance);
118
119         newClaim.claimedValue = uint128(claimedValue);
120         newClaim.fromBlock = uint128(block.number);
121
122         require(generateTokens(_owner, newClaim.claimedValue));
123
124         emit Claimed(_owner, newClaim.claimedValue);
125       }
126     }
```

POSController.sol

doClaim 함수에서 *claimRate* 가 0 보다 클 경우 *claim* 에 대한 정보를 기록합니다. POSController:19 에서 보듯이 *Claim struct* 에서 *claimedValue* 는 *uint128* 입니다. 하지만 POSController:117 에서 계산되는 *claimedValue* 는 *uint256* 입니다. 일반적인 경우에는 한번의 claim 으로 큰 숫자의 *claimedValue* 가 생성되지 않을 것이지만, *posInterval* 을 매우 여러번 거친 뒤 *claim* 을 하게되면 *claimedValue* 가 2^128 보다 클 수 있습니다. 그러므로 안전하게 코딩하기 위해서는 *struct* 의 *claimedValue* 를 *uint256* 으로 변경하는 것을 추천합니다.

3. POSController.sol 에서 *doClaim* 을 할 때 *owner* 의 *balance* 가 없는 경우 불필요한 로직을 수행하지 않는 것을 추천합니다.

"Issue 2" 에서 살펴본 *doClaim* 에서 114번째 줄을 보면 *owner* 주소의 *balance* 를 가져오는 것을 확인할 수 있습니다. 이 때 *balance* 가 0 이라면 114 번째 줄 이후의 함수 실행은 의미가 없습니다. 따라서 *balance* 가 0 일 경우에 추가 토큰 생성하는 로직을 실행하지 않게 함으로써 불필요한 연산을 줄일 수 있습니다. 이는 가스 소모량을 줄이는데도 도움이 됩니다.

4. POSController.sol 의 *setRate*, *setInterval* 에서 *block.number* 를 같이 기록하는 것을 추천합니다.

```
59    /* External */
60    function setRate(uint256 _newRate) external onlyOwner {
61      require(_newRate != 0);
62      posRate = _newRate;
63    }
```

POSController.sol

claimToken 을 실행할 경우 생성할 수 있는 토큰이 있지만 *claimToken* 을 호출하지 않고 있을 수 있습니다. 이런 상황에서 *setRate* 함수와 *setInterval* 함수를 이용해서 *posRate* 와 *posInterval* 을 변경하게 된다면 claimToken 으로 생성되는 토큰의 양이 달라질 수 있습니다. 따라서 *posRate*, *posInterval* 를 변경하기 전에 모든 토큰 홀더들에게 분배될 수 있는 토큰을 생성하는 것이 좋습니다. 하지만 이 방법은 *posRate*, *posInterval* 을 변경할 때 마다 가스비가 많이 들기 때문에 현실적으로 어려울 수 있습니다. 따라서 *posRate*, *posInterval* 을 변경할 때 *block.number* 를 같이 기록하고 *claimToken* 에서 추가적으로 생성되는 토큰을 계산할 때 *posRate*, *posInterval* 와 함께 기록한 *block.number* 를 이용해서 올바른 이자를 주는 것이 바람직합니다.

5. BalanceUpdateMiniMeToken.sol 에 불필요한 연산이 존재합니다.

```
9     /// @dev Override doTransfer function. only modified parts are documented.
10    function doTransfer(address _from, address _to, uint _amount
11    ) internal {
12
13      if (_amount == 0) {
14        Transfer(_from, _to, _amount);
15        return;
16      }
17
18      require(parentSnapShotBlock < block.number);
19      require((_to != 0) && (_to != address(this)));
20
21      var previousBalanceFrom = balanceOfAt(_from, block.number);
22      require(previousBalanceFrom >= _amount);
23
24      if (isContract(controller)) {
25        require(TokenController(controller).onTransfer(_from, _to, _amount));
26
27        // update balance
28        previousBalanceFrom = balanceOfAt(_from, block.number);
29        require(previousBalanceFrom >= _amount);
30      }
31
32      updateValueAtNow(balances[_from], previousBalanceFrom - _amount);
33
34      var previousBalanceTo = balanceOfAt(_to, block.number);
35      require(previousBalanceTo + _amount >= previousBalanceTo); // Check for overflow
36      updateValueAtNow(balances[_to], previousBalanceTo + _amount);
37
38      Transfer(_from, _to, _amount);
39    }
40  }
```

BalanceUpdatableMiniMeToken.sol

BalanceUpdateMinimeToken.sol:25 에서 *onTransfer* 함수가 호출 될 때 *POSController* 의 *claimTokens* 함수가 호출됩니다. 따라서 *balance* 가 바뀔 수 있기 때문에 28, 29 번째 줄이 필요합니다. 하지만 이 경우 21, 22 번째 줄에 대한 연산을 미리할 필요가 없습니다. 따라서 21, 22 번째 줄을 삭제하고 28, 29 번째 줄을 24번째 줄의 *isContract* 괄호 밖으로 옮겨서 불필요한 연산을 줄이는 것을 추천합니다.

6. BalanceUpdatableMiniMeToken.sol 에서 *var* 자료형을 사용하지 않는 것을 권장합니다.

BalanceUpdatableMiniMeToken.sol:21, BalanceUpdatableMiniMeToken.sol:34 에 사용되는 *var* 자료형은 solidity 0.4.20 부터 deprecate(https://github.com/ethereum/solidity/releases/tag/v0.4.20) 되었고 원래 자료형이 무엇인지 유추하기 어렵습니다. 따라서 *var* 대신 *uin256* 을 쓰는 것을 권장합니다.

7. POSController.sol 의 *onApprove* 함수에서 *claimTokens* 를 호출할 필요가 없습니다.

```
92    /// @notice onApprove implements MiniMeToken Controller's onApprove
93    function onApprove(address _owner, address _spender, uint _amount) public returns(bool) {
94      claimTokens(_owner);
95      return true;
96    }
```

POSController.sol

POSController.sol:93 의 *onApprove* 함수는 MiniMeToken.sol:229 에서 처럼 Token 컨트랙트의 *approve* 함수가 호출될 때 호출됩니다. *approve* 는 실제 *balance* 가 조정되는 것이 아니라 token 을 전달할 수 있는 권한을 주는 것이기 때문에 현재의 *balance* 와 무관합니다. 따라서 *claimTokens* 를 호출할 필요가 없습니다. POSController.sol:93 처럼 *_owner* 에 대해서는 *claimTokens* 를 호출하고 *_spender* 에 대해서는 호출하지 않으면 더 헷갈릴 수 있습니다. 둘다 호출을 생략하거나 *approve* 와 무관하지만 *claimTokens* 를 하고싶다면 *_owner*, *_spender* 모두 호출할 것을 권장합니다.

05. DISCLAIMER

해당 리포트는 투자에 대한 조언, 비즈니스 모델의 적합성, 버그 없이 안전한 코드를 보증하지 않습니다. 해당 리포트는 알려진 기술 문제들에 대한 논의의 목적으로만 사용됩니다. 리포트에 기술된 문제 외에도 이더리움, 솔리디티 상의 결함, 발견되지 않은 문제들이 있을 수 있습니다. 안전한 스마트 컨트랙트를 작성하기 위해서는 발견된 문제들에 대한 수정과 충분한 테스트가 필요합니다.