BEOSIN
Blockchain Security

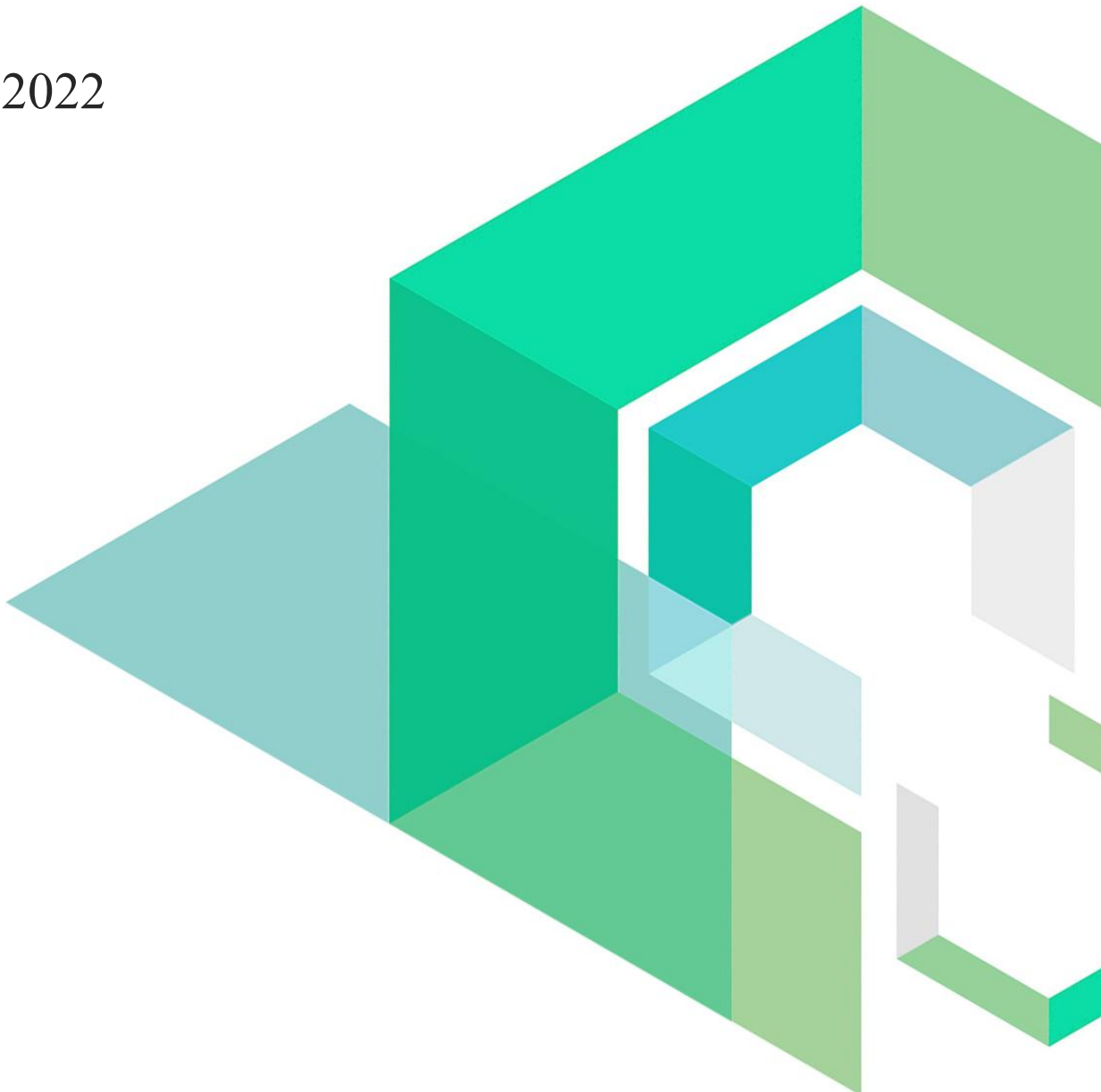
Polygon

Smart Contract Security Audit

V1.0

No. 202206151130

Jun 15th, 2022



Contents

Summary of audit results

After auditing, 2 High-risk, 1 Medium-risk, 3 Low-risk and 2 Info items were identified in the Polygon project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project:

*Notes:

- **Risk Description:**

1. **Centralization risk**

The owner can set the operator address, and both the owner and operator addresses can modify key parameters in the contract. For some parameters, only the owner has the permission to modify. There may be some centralization risk.

2. **Risk of insufficient gas**

Multiple for loops are used in many places in the contract. If there are too many loops, the related function calls may fail.

3. **Signature reuse risk**

In the *_checkUnstakeFeeSignature* function, the nonce is not used to limit the number of times the signature is used when verifying the signature data, which may cause the signature data to be used multiple times. If the fee changes dynamically, it may cause users to withdraw with a lower rate multiple times.

- **Project Description:**

1. Basic Token Information

Token name	Ankr MATIC Reward Earning Bond
Token symbol	aMATICb
Decimals	18
Pre-mint	0
Total supply	Initial supply is 0 (Mintable, burnable)
Token type	ERC-20

Table 1 aMATICb token info

Token name	Ankr MATIC Reward Bearing Certificate
Token symbol	aMATICc
Decimals	18
Pre-mint	0
Total supply	Initial supply is 0 (Mintable, burnable)
Token type	ERC-20

Table 2 aMATICc token info

2. Business overview

The Polygon project contains a token contract and two business contracts. In the token contract, the number of shares is recorded inside the contract, and what the user queries is the number of bonds. Shares and bonds are converted according to a certain ratio (the ratio can be arbitrarily modified by the owner or operator address). Users can stake Matic tokens in the PolygonPool contract to obtain aMATICb tokens, and the aMATICb tokens and aMATICc tokens are interchangeable on a 1:1 ratio. When the user withdraws the matic tokens staked in the PolygonPool contract, he needs to first apply for the withdrawal through the data signed by the notary address, and then the operator address calls the serveClaims function to send the tokens to the user. And when the user withdraws, a certain amount of ANKR Token will be charged as a handling fee.

1 Overview

1.1 Project Overview

Project Name	Polygon	
Platform	Ethereum	
File Hash (SHA256)	aMATICb.sol	d62eaa020483e77b7f221df649d2a5328428d6d6425ea3bdd9c14f4a9afee517(Initial) 59a3d5e421b90849b93581e811f81fac3db384c61bdd217f530449721aa379c9 (Final)
	aMATICc.sol	4f49008c971bd4165011a742862208b7a96e622d6f24592d3c3250c7519bf8ab (Initial) 5add76bdb752147e69adf66c26023635e21b7d69290a3e68849a04f2b8cc193b (Final)
	PolygonPool.sol	91b63cf8abffc7448ada8a9d98d136f1866d436545d2a28cac3840688c1226c2 (Initial) f2cf0cd8c40c8f4da75dc5e6228825a5c837268ddce48fad87da438724f676d6 (Final)

1.2 Audit Overview

Audit work duration: April 24 , 2022 – June 15, 2022

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

Audit team: Beosin Technology Co. Ltd.

2 Findings

Index	Risk description	Severity level	Status
Polygon-1	Operator and owner have a high authority	High	Acknowledged
Polygon-2	Owner has a large permission	High	Fixed
Polygon-3	*ClaimToIntermediary* function implementation problem	Medium	Fixed
Polygon-4	Centralization risk	Low	Acknowledged
Polygon-5	Signature reuse risk	Low	Acknowledged
Polygon-6	Risk of insufficient gas	Low	Acknowledged
Polygon-7	The corresponding event is not triggered	Info	Fixed
Polygon-8	Redundant code	Info	Acknowledged

Risk Details Description:

1. Polygon-1 is not fixed and may cause users to be unable to withdraw assets.

2. Polygon-4 is not fixed and may cause a potential centralization risk.

3. Polygon-5 is not fixed and may cause users to withdraw with a lower fee multiple times.

4. Polygon-6 is not fixed and may cause the related function call to fail when there are too many loops.

5. Polygon-8 is not fixed but does not cause security issues.

[Polygon-1] Operator and owner have a high authority

Severity Level	**High**
Type	Business Security
Lines	PolygonPool_R2.sol#L240-326
Description	In the PolygonPool_R2 contract, the withdrawal request requires the signature of the notary address after the user stakes, and after the request is initiated, the operator address operation is required to send the token to the user. If the notary address and operator address do not operate as expected, users may not be able to withdraw tokens staked in the contract.

```
240     function unstake(uint256 amount, uint256 fee, uint256 useBeforeBlock, bytes memory signature) override external nonReentrant {
241         uint256 shares = IinternetBond_R1(_bondContract).balanceToShares(amount);
242         _unstake(msg.sender, amount, shares, fee, useBeforeBlock, signature, true);
243     }
244
245     function unstakeBonds(uint256 amount, uint256 fee, uint256 useBeforeBlock, bytes memory signature) override external nonReentrant {
246         uint256 shares = IinternetBond_R1(_bondContract).balanceToShares(amount);
247         _unstake(msg.sender, amount, shares, fee, useBeforeBlock, signature, true);
248     }
249
250     function unstakeCerts(uint256 shares, uint256 fee, uint256 useBeforeBlock, bytes memory signature) override external nonReentrant {
251         uint256 amount = IinternetBond_R1(_bondContract).sharesToBalance(shares);
252         IinternetBond_R1(_bondContract).lockSharesFor(msg.sender, shares);
253         _unstake(msg.sender, amount, shares, fee, useBeforeBlock, signature, false);
254     }
255
256     function _unstake(address staker, uint256 amount, uint256 shares, uint256 fee, uint256 useBeforeBlock, bytes memory signature, bool isRebasing) internal {
257         require(IERC20Upgradeable(_bondContract).balanceOf(staker) >= amount, "cannot claim more than have on address");
258         require(block.number < useBeforeBlock, "fee approval expired");
259         require(
260             _checkUnstakeFeeSignature(fee, useBeforeBlock, staker, signature),
261             "Invalid unstake fee signature"
262         );
263         require(_ankrToken.transferFrom(staker, _feeCollector, fee), "could not transfer unstake fee");
264         _pendingClaimers.push(staker);
265         _pendingClaimAmounts.push(amount);
266         IinternetBond_R1(_bondContract).lockForDelayedBurn(staker, amount);
267         emit MaticClaimPending(staker, amount);
268         emit TokensBurned(staker, amount, shares, fee, isRebasing);
269     }
```

Figure 1 Source code of related functions

```
271     function serveClaims(uint256 amountToUse, address payable residueAddress, uint256 minThreshold) public onlyOperator payable {
272         address[] memory claimers = new address[](_pendingClaimers.length.sub(_pendingMaticClaimGap));
273         uint256[] memory amounts = new uint256[](_pendingClaimers.length.sub(_pendingMaticClaimGap));
274         uint256 availableAmount = _maticToken.balanceOf(address(this));
275         availableAmount = availableAmount.sub(_getTotalPendingStakes());
276         require(amountToUse <= availableAmount, "not enough MATIC tokens to serve claims");
277         if (amountToUse > 0) {
278             availableAmount = amountToUse;
279         }
280         uint256 j = 0;
281         uint256 gaps = 0;
282         uint256 i = 0;
283         for (i = _pendingMaticClaimGap; i < _pendingClaimers.length; i++) {
284             /* if the number of tokens left is less than threshold do not try to serve the claims */
285             if (availableAmount < minThreshold) {
286                 break;
287             }
288             address claimer = _pendingClaimers[i];
289             uint256 amount = _pendingClaimAmounts[i];
290             /* we might have gaps lets just skip them (we shrink them on full claim) */
291             if (claimer == address(0) || amount == 0) {
292                 gaps++;
293                 continue;
294             }
295             if (availableAmount < amount) {
296                 break;
297             }
298             claimers[j] = claimer;
299             amounts[j] = amount;
300             address payable wallet = payable(address(claimer));
301             _maticToken.transfer(wallet, amount);
302             availableAmount = availableAmount.sub(amount);
303             j++;
304             IinternetBond_R1(_bondContract).commitDelayedBurn(claimer, amount);
305             delete _pendingClaimAmounts[i];
306             delete _pendingClaimers[i];
307             /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim '
308             gaps++;
309         }
310         _pendingMaticClaimGap = _pendingMaticClaimGap.add(gaps);
311         uint256 missing = 0;
312         for (i = _pendingMaticClaimGap; i < _pendingClaimers.length; i++) {
313             missing = missing.add(_pendingClaimAmounts[i]);
314         }
315         /* Send event with results */
316         if (availableAmount > 0) {
317             _maticToken.transfer(residueAddress, availableAmount);
318         }
319         /* decrease arrays */
320         uint256 removeCells = claimers.length.sub(j);
321         if (removeCells > 0) {
322             assembly {mstore(claimers, sub(mload(claimers), removeCells))}
323             assembly {mstore(amounts, sub(mload(amounts), removeCells))}
324         }
325         emit ClaimsServed(claimers, amounts, missing);
326     }
```

Figure 2 Source code of *serveClaims* function

Recommendations	It is recommended to remove the restriction on users' withdrawal of principal.
Status	Acknowledged. The project party confirmed the logic of this part of the code.

[Polygon-2] Owner has a high authority

Severity Level	**High**
Type	Business Security
Lines	aMATICb_R3.sol#L108-125, 214-222
Description	In the aMATICb_R3 contract, the owner can call *mintBonds* and *mint* functions to mint tokens at will, and call *burn* and *commitDelayedBurn* functions to burn tokens at any address.

```
108    function mintBonds(address account, uint256 amount) public override onlyBondMinter {
109        uint256 shares = _bondsToShares(amount);
110        _mint(account, shares);
111        emit Transfer(address(0), account, _sharesToBonds(shares));
112    }
113
114    function mint(address account, uint256 shares) public onlyMinter {
115        _lockedShares = _lockedShares.sub(int256(shares));
116        _mint(account, shares);
117        emit Transfer(address(0), account, _sharesToBonds(shares));
118    }
119
120    function burn(address account, uint256 amount) public override onlyMinter {
121        uint256 shares = _bondsToShares(amount);
122        _lockedShares = _lockedShares.add(int256(shares));
123        _burn(account, shares);
124        emit Transfer(account, address(0), _sharesToBonds(shares));
125    }
```

Figure 3 Source code of related function

```
214    modifier onlyMinter() {
215        require(msg.sender == _crossChainBridge, "Minter: not allowed");
216        _;
217    }
218
219    modifier onlyBondMinter() {
220        require(msg.sender == _polygonPool, "Minter: not allowed");
221        _;
222    }
```

Figure 4 Source code of *onlyMinter* and *onlyBondMinter* modifiers (Unfixed)

Recommendations	It is recommended to cancel the corresponding permissions of owner in the modifiers.
Status	Fixed.

```
214     modifier onlyMinter() {
215         require(msg.sender == _crossChainBridge, "Minter: not allowed");
216         _;
217     }
218
219     modifier onlyBondMinter() {
220         require(msg.sender == _polygonPool, "Minter: not allowed");
221         _;
222     }
```

Figure 5 Source code of *onlyMinter* and *onlyBondMinter* modifiers (Fixed)

[Polygon-3] *ClaimToIntermediary* function implementation problem

Severity Level	Medium
Type	Business Security
Lines	PolygonPool_R2.sol#L121-166
Description	The incorrect use of msg.value in the *claimToIntermediary* function of the PolygonPool contract may cause the function call to fail and ETH to be locked in the contract.

```
121  function claimToIntermediary(address payable intermediary, uint256 threshold) public onlyOperator payable {
122      address[] memory stakers = new address[](_pendingStakers.length.sub(_pendingGap));
123      uint256[] memory amounts = new uint256[](_pendingStakers.length.sub(_pendingGap));
124      uint256 total = 0;
125      uint256 j = 0;
126      uint256 gaps = 0;
127      uint256 i = 0;
128      for (i = _pendingGap; i < _pendingStakers.length; i++) {
129          /* if total exceeds threshold then we can't proceed stakes anymore (don't move this check to the end of scope) */
130          if (total >= threshold) {
131              break;
132          }
133          address staker = _pendingStakers[i];
134          uint256 amount = _pendingUserStakes[staker];
135          /* we might have gaps lets just skip them (we shrink them on full claim) */
136          if (staker == address(0) || amount == 0) {
137              gaps++;
138              continue;
139          }
140          /* if stake amount with current total exceeds threshold then split it */
141          if (total.add(amount) > threshold) {
142              amount = threshold.sub(total);
143          }
144          stakers[j] = staker;
145          amounts[j] = amount;
146          total = total.add(amount);
147          j++;
148          /* lets release pending stakes only if amount is zero */
149          _pendingUserStakes[staker] = _pendingUserStakes[staker].sub(amount);
150          if (_pendingUserStakes[staker] == 0) {
151              delete _pendingStakers[i];
152              /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim */
153              gaps++;
154          }
155      }
156      _pendingGap = _pendingGap.add(gaps);
157      /* claim funds to intermediary */
158      _maticToken.transfer(intermediary, total.add(msg.value));
159      /* decrease arrays */
160      uint256 removeCells = stakers.length.sub(j);
161      if (removeCells > 0) {
162          assembly {mstore(stakers, sub(mload(stakers), removeCells))}
163          assembly {mstore(amounts, sub(mload(amounts), removeCells))}
164      }
```

Figure 6 Source code of *claimToIntermediary* function (Unfixed)

Recommendations	It is recommended to remove the payable modifier of the *claimToIntermediary* function and the msg.value in the *transfer* function parameter.
Status	Fixed.

```
129     function claimToIntermediary(address payable intermediary, uint256 threshold) public onlyOperator {
130         address[] memory stakers = new address[](_pendingStakers.length.sub(_pendingGap));
131         uint256[] memory amounts = new uint256[](_pendingStakers.length.sub(_pendingGap));
132         uint256 total = 0;
133         uint256 j = 0;
134         uint256 gaps = 0;
135         uint256 i = 0;
136         for (i = _pendingGap; i < _pendingStakers.length; i++) {
137             /* if total exceeds threshold then we can't proceed stakes anymore (don't move this check to the end of scope) */
138             if (total >= threshold) {
139                 break;
140             }
141             address staker = _pendingStakers[i];
142             uint256 amount = _pendingUserStakes[staker];
143             /* we might have gaps lets just skip them (we shrink them on full claim) */
144             if (staker == address(0) || amount == 0) {
145                 gaps++;
146                 continue;
147             }
148             /* if stake amount with current total exceeds threshold then split it */
149             if (total.add(amount) > threshold) {
150                 amount = threshold.sub(total);
151             }
152             stakers[j] = staker;
153             amounts[j] = amount;
154             total = total.add(amount);
155             j++;
156             /* lets release pending stakes only if amount is zero */
157             _pendingUserStakes[staker] = _pendingUserStakes[staker].sub(amount);
158             if (_pendingUserStakes[staker] == 0) {
159                 delete _pendingStakers[i];
160                 /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim */
161                 gaps++;
162             }
163         }
164         _pendingGap = _pendingGap.add(gaps);
165         /* claim funds to intermediary */
166         require(_maticToken.transfer(intermediary, total), "matic not sent");
167         /* decrease arrays */
168         uint256 removeCells = stakers.length.sub(j);
169         if (removeCells > 0) {
170             assembly {mstore(stakers, sub(mload(stakers), removeCells))}
171             assembly {mstore(amounts, sub(mload(amounts), removeCells))}
172         }
173         emit IntermediaryClaimed(stakers, amounts, intermediary, total);
174     }
```

Figure 7 Source code of *claimToIntermediary* function (Fixed)

[Polygon-4] Centralization risk

Severity Level	**Low**
Type	Business Security
Lines	aMATICb _R3.sol#L281-291, L226-236 PolygonPool _R2.sol#L188-198, L332-346, L352-366
Description	The owner of the aMATICb contract can call functions such as *changeOperator*, *changePolygonPool*, *changeCrossChainBridge* and other functions to modify contract-related parameters. The owner authority in the PolygonPool contract can call the function *changeBondContract* , *setNotary* , *setAnkrTokenAddress* , and *setMinimumStake* to modify the contract-related parameters.

```
226    function changeOperator(address operator) public onlyOwner {
227        _operator = operator;
228    }
229
230    function changePolygonPool(address polygonPool) public onlyOwner {
231        _polygonPool = polygonPool;
232    }
233
234    function changeCrossChainBridge(address crossChainBridge) public onlyOwner {
235        _crossChainBridge = crossChainBridge;
236    }
```

Figure 8 Source code of related functions

```
352    function setMinimumStake(uint256 minStake) public onlyOperator {
353        _minimumStake = minStake;
354    }
355
356    function setFeeCollector(address feeCollector) public onlyOwner {
357        _feeCollector = feeCollector;
358    }
359
360    function setNotary(address notary) public onlyOwner {
361        _notary = notary;
362    }
363
364    function setAnkrTokenAddress(IERC20Upgradeable ankrToken) public onlyOwner {
365        _ankrToken = ankrToken;
366    }
```

Figure 9 Source code of related functions

Recommendations	It is recommended to use multi-signature wallet, TimeLock contract, DAO, etc. as the contract owner.
Status	Acknowledged.

[Polygon-5] Signature reuse risk

Severity Level	**Low**
Type	Business Security
Lines	PolygonPool_R3.sol#L382-387
Description	In the *_checkUnstakeFeeSignature* function, the nonce is not used to limit the number of times the signature is used when verifying the signature data, which may cause the signature data to be used multiple times. If the fee changes dynamically, it may cause users to withdraw with a lower fee multiple times.

```
382     function _checkUnstakeFeeSignature(
383         uint256 fee, uint256 useBeforeBlock, address staker, bytes memory signature
384     ) public view returns (bool) {
385         bytes32 payloadHash = keccak256(abi.encode(currentChain(), address(this), fee, useBeforeBlock, staker));
386         return ECDSAUpgradeable.recover(payloadHash, signature) == _notary;
387     }
```

Figure 9 Source code of *_checkUnstakeFeeSignature* functions

Recommendations	It is recommended to add a nonce to the signature data.
Status	Acknowledged. The project party confirmed the logic of this part of the code.

[Polygon-6] Risk of insufficient gas

Severity Level	**Low**
Type	Business Security
Lines	PolygonPool _R2.sol#L281-336, L397-406, L342-352, L220-234, L101-114, L129-174, L210-218, L236-248
Description	In the PolygonPool contract, *calcPendingClaimGap*, *getRawPendingStakes*, *getPendingClaims*, *getPendingStakes*, *claimToIntermediary*, *pendingMaticClaimsOf*, *getRawPendingClaims*, *serveClaims* and other functions use for loops. If the length of the related array is too large, the call may fail due to insufficient gas.

```
281  function serveClaims(uint256 amountToUse, address payable residueAddress, uint256 minThreshold) public onlyOperator payable {
282      address[] memory claimers = new address[](_pendingClaimers.length.sub(_pendingMaticClaimGap));
283      uint256[] memory amounts = new uint256[](_pendingClaimers.length.sub(_pendingMaticClaimGap));
284      uint256 availableAmount = _maticToken.balanceOf(address(this));
285      availableAmount = availableAmount.sub(_getTotalPendingStakes());
286      require(amountToUse <= availableAmount, "not enough MATIC tokens to serve claims");
287      if (amountToUse > 0) {
288          availableAmount = amountToUse;
289      }
290      uint256 j = 0;
291      uint256 gaps = 0;
292      uint256 i = 0;
293      for (i = _pendingMaticClaimGap; i < _pendingClaimers.length; i++) {
294          /* if the number of tokens left is less than threshold do not try to serve the claims */
295          if (availableAmount < minThreshold) {
296              break;
297          }
298          address claimer = _pendingClaimers[i];
299          uint256 amount = _pendingClaimAmounts[i];
300          /* we might have gaps lets just skip them (we shrink them on full claim) */
301          if (claimer == address(0) || amount == 0) {
302              gaps++;
303              continue;
304          }
305          if (availableAmount < amount) {
306              break;
307          }
308          claimers[j] = claimer;
309          amounts[j] = amount;
310          address payable wallet = payable(address(claimer));
311          require(_maticToken.transfer(wallet, amount), "cannot send matic to claimer");
312          availableAmount = availableAmount.sub(amount);
313          j++;
314          IinternetBond_R1(_bondContract).commitDelayedBurn(claimer, amount);
315          delete _pendingClaimAmounts[i];
316          delete _pendingClaimers[i];
317          /* when we delete items from array we generate new gap, lets remember how many gaps we did to skip them in next claim */
318          gaps++;
319      }
320      _pendingMaticClaimGap = _pendingMaticClaimGap.add(gaps);
321      uint256 missing = 0;
322      for (i = _pendingMaticClaimGap; i < _pendingClaimers.length; i++) {
323          missing = missing.add(_pendingClaimAmounts[i]);
324      }
```

Figure 11 Source code of related functions

Recommendations	It is recommended to use a separate variable to store the total amount of stakes and withdrawals to avoid traversing the array and consuming too much gas. When getting missing data in the *serveClaims* function, use the total amount minus the number that has been claimed instead of traversing the array to calculate.
Status	Acknowledged.

[Polygon-7] The corresponding event is not triggered

Severity Level	Info
Type	Coding Conventions
Lines	PolygonPool_R2.sol#L352-366, L192-198
Description	Functions such as *changeOperator*, *changeBondContract*, *setFeeCollector*, *setNotar*, and *setAnkrTokenAddress* in the *PolygonPool* contract are not triggered when they are called.

```
352     function setMinimumStake(uint256 minStake) public onlyOperator {
353         _minimumStake = minStake;
354     }
355
356     function setFeeCollector(address feeCollector) public onlyOwner {
357         _feeCollector = feeCollector;
358     }
359
360     function setNotary(address notary) public onlyOwner {
361         _notary = notary;
362     }
363
364     function setAnkrTokenAddress(IERC20Upgradeable ankrToken) public onlyOwner {
365         _ankrToken = ankrToken;
366     }
```

Figure 10 Source code of related functions (Unfixed)

```
192     function changeOperator(address operator) public onlyOwner {
193         _operator = operator;
194     }
195
196     function changeBondContract(address bondContract) public onlyOwner {
197         _bondContract = bondContract;
198     }
```

Figure 11 Source code of related functions (Unfixed)

Recommendations	It is recommended to trigger the corresponding event.
Status	Fixed.

```
362     function setMinimumStake(uint256 minStake) public onlyOperator {
363         _minimumStake = minStake;
364         emit MinimumStakeChanged(minStake);
365     }
366
367     function setFeeCollector(address feeCollector) public onlyOwner {
368         _feeCollector = feeCollector;
369         emit FeeCollectorChanged(feeCollector);
370     }
371
372     function setNotary(address notary) public onlyOwner {
373         _notary = notary;
374         emit NotaryChanged(notary);
375     }
376
377     function setAnkrTokenAddress(IERC20Upgradeable ankrToken) public onlyOwner {
378         _ankrToken = ankrToken;
379         emit AnkrTokenAddressChanged(address(ankrToken));
380     }
```

Figure 12 Source code of related functions (Fixed)

```
200     function changeOperator(address operator) public onlyOwner {
201         _operator = operator;
202         emit OperatorChanged(operator);
203     }
204
205     function changeBondContract(address bondContract) public onlyOwner {
206         _bondContract = bondContract;
207         emit BondContractChanged(bondContract);
208     }
```

Figure 13 Source code of related functions (Fixed)

[Polygon-8] Redundant code

Severity Level	Info
Type	Coding Conventions
Lines	PolygonPool_R2.sol#L15-19, L39
Description	The PausableUpgradeable module is inherited in the PolygonPool contract, but the pause function is not implemented in the main contract. The variable _collectedFee is not used in the contract.

```
15  contract PolygonPool_R2 is PausableUpgradeable, ReentrancyGuardUpgradeable, OwnableUpgradeable, IGlobalPool_R1 {
16
17      using SafeMathUpgradeable for uint256;
18      using MathUpgradeable for uint256;
19
```

Figure 14 Source code of Polygon contract

```
35      mapping(address => uint256) private _pendingUserStakes;
36      address[] private _pendingStakers;
37      address private _operator;
38      address private _notary;
39      uint256 private _collectedFee;
40      uint256 private _minimumStake;
41      address private _bondContract;
```

Figure 15 Unused variable _collectedFee

Recommendations	It is recommended to delete the relevant code.
Status	Acknowledged.

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 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.

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

3.1.2 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.

3.1.5 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.

3.2 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.

3.3 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.

3.4 About BEOSIN

Affiliated to BEOSIN Technology Pte. Ltd., 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.

BEOSIN
Blockchain Security

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com