SMART CONTRACT AUDIT REPORT

for

MATIC POS PORTAL

Prepared By: Yiqun Chen

PeckShield

July 30, 2021

Document Properties

Client	Polygon
Title	Smart Contract Audit Report
Target	Polygon
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 26, 2021	Xuxian Jiang	Release Candidate #1
0.2	June 14, 2021	Xuxian Jiang	Additional Findings #1
0.1	May 31, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Matic PoS portal contracts`, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Polygon

`Polygon`, previously `Matic Network`, is a layer-2 scaling solution that achieves scale by utilizing sidechains for off-chain computation while ensuring asset security using the Plasma framework and a decentralized network of proof-of-stake validators. The audited system, `Matic PoS portal contracts`, contains a set of smart contracts that power the `proof-of-stake (PoS)` based bridge mechanism for `Polygon`. And the system is designed to allow for asset-transfers between the `Polygon` and `Ethereum` by effectively acting as a cross-chain bridge, including `ERC20`, `ERC721`, `ERC1155`, and other token standards when required.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Polygon

Item	Description
Target	Polygon
Website	https://matic.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/maticnetwork/pos-portal.git (d062711)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/maticnetwork/pos-portal.git (1ed233f2)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Matic PoS portal contracts`. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	5	■■■■■
Informational	1	■
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Ether Transfer	Coding Practices	Fixed
PVE-002	Low	Improved Gas Efficiency in RLPReader::copy()	Coding Practices	Fixed
PVE-003	Low	Improved Sanity Checks For System/Function Parameters	Coding Practices	Fixed
PVE-004	Low	Suggested Versioned Initialization	Coding Practices	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	
PVE-007	Informational	Single Common Allowance Target For All Predicates	Business Logic	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Ether Transfers

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: EtherPredicate
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

As mentioned earlier, Matic PoS portal contracts are designed to allow for asset-transfers between the Polygon and Ethereum by effectively acting as a cross-chain bridge, including ERC20, ERC721, ERC1155 , and other token standards. While examining the transfers of native assets, we notice the current implementation can be improved.

To elaborate, we show below the exitTokens() function from the EtherPredicate contract. It is suggested to support Ether-related cross-chain transfer. We observe that the Solidity function transfer() is used (line 92 in the code snippet below). However, as described in [1], when the recipient happens to be a contract that implements a callback function containing EVM instructions such as SLOAD, the 2300 gas supplied with transfer() might not be sufficient, leading to an out-of-gas error.

```
60      /**
61       * @notice Validates log signature, from and to address
62       * then sends the correct amount to withdrawer
63       * callable only by manager
64       * @param log Valid ERC20 burn log from child chain
65       */
66      function exitTokens(
67          address,
68          address,
69          bytes memory log
70      )
71          public
```

```
        override
        only(MANAGER_ROLE)
    {
        RLPReader.RLPItem[] memory logRLPList = log.toRlpItem().toList();
        RLPReader.RLPItem[] memory logTopicRLPList = logRLPList[1].toList(); // topics

        require(
            bytes32(logTopicRLPList[0].toUint()) == TRANSFER_EVENT_SIG, // topic0 is
                event sig
            "EtherPredicate: INVALID_SIGNATURE"
        );

        address withdrawer = address(logTopicRLPList[1].toUint()); // topic1 is from
            address

        require(
            address(logTopicRLPList[2].toUint()) == address(0), // topic2 is to address
            "EtherPredicate: INVALID_RECEIVER"
        );

        emit ExitedEther(withdrawer, logRLPList[2].toUint());

        payable(withdrawer).transfer(logRLPList[2].toUint());
    }
```

Listing 3.1: `EtherPredicate::exitTokens()`

As suggested in [1], we may consider avoiding the direct use of Solidity's `transfer()` as well. Meanwhile, we need to exercise extra caution during the use of `call()` as it may lead to side effects such as `re-entrancy` and gas token vulnerabilities. In other words, we need to specify the maximum allowed gas amount when making the (untrusted) external `call()`.

Recommendation When transferring `ETH`, it is suggested to replace the Solidity function `transfer()` with `call()`.

Status The issue has been fixed by this commit: `3c85192`.

3.2 Improved Gas Efficiency in RLPReader::copy()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RLPReader`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

The current code base has integrated a third-party library, i,e., `RLPReader`, that allows for efficient and robust parsing of given `RLP`-encoded log information (in `bytes`). While analyzing the commit history of this third-party library, we notice that the used library version is based on a specific commit hash submitted on August 22, 2019 while the latest version of the same library is committed on April 4, 2021.

Our analysis with intermediate versions show they pose no security risk to current library code. However, we do observe the performance optimization in the latest code. To illustrate, we show below the `copy()` that basically copies from the given source to the destination up to the given amount of bytes. The final copy of left-over bytes (lines 255 – 260) can be avoided when the resulting `len=0`.

```
237     function copy(
238         uint256 src,
239         uint256 dest,
240         uint256 len
241     ) private pure {
242         if (len == 0) return;

244         // copy as many word sizes as possible
245         for (; len >= WORD_SIZE; len -= WORD_SIZE) {
246             assembly {
247                 mstore(dest, mload(src))
248             }

250             src += WORD_SIZE;
251             dest += WORD_SIZE;
252         }

254         // left over bytes. Mask is used to remove unwanted bytes from the word
255         uint256 mask = 256**(WORD_SIZE - len) - 1;
256         assembly {
257             let srcpart := and(mload(src), not(mask)) // zero out src
258             let destpart := and(mload(dest), mask) // retrieve the bytes
259             mstore(dest, or(destpart, srcpart))
260         }
261     }
```

Listing 3.2: `RLPReader::copy()`

For better maintenance and compatibility, we strongly suggest the use of the latest (stable) library version.

Recommendation Upgrade the `RLPReader` library to its latest version.

Status The issue has been fixed by this commit: `3388ea3`.

3.3 Improved Sanity Checks For System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. `Matic PoS portal contracts` are no exception. Specifically, if we examine the `BaseRootTunnel` contract, it has a public `receiveMessage()` function that handles the asset-transfer message from child chain to root chain. Naturally, it needs to validate by proof so that the transaction actually happened on child chain.

Specifically, the asset-transfer message is processed by the following routine `_validateAndExtractMessage()`. And this function can be improved by properly validating the given message, including the input has enough bytes. Note that a similar issue is also present in another routine, i.e., `RootChainManager::exit()`.

```
    function _validateAndExtractMessage(bytes memory inputData) internal returns (bytes
        memory) {
        RLPReader.RLPItem[] memory inputDataRLPList = inputData
            .toRlpItem()
            .toList();

        // checking if exit has already been processed
        // unique exit is identified using hash of (blockNumber, branchMask,
            receiptLogIndex)
        bytes32 exitHash = keccak256(
            abi.encodePacked(
                inputDataRLPList[2].toUint(), // blockNumber
                // first 2 nibbles are dropped while generating nibble array
                // this allows branch masks that are valid but bypass exitHash check (
                    changing first 2 nibbles only)
                // so converting to nibble array and then hashing it
                MerklePatriciaProof._getNibbleArray(inputDataRLPList[8].toBytes()), //
                    branchMask
```

```
                inputDataRLPList[9].toUint() // receiptLogIndex
            )
        );
        require(
            processedExits[exitHash] == false,
            "RootTunnel: EXIT_ALREADY_PROCESSED"
        );
        processedExits[exitHash] = true;

        RLPReader.RLPItem[] memory receiptRLPList = inputDataRLPList[6]
            .toBytes()
            .toRlpItem()
            .toList();
        RLPReader.RLPItem memory logRLP = receiptRLPList[3]
            .toList()[
                inputDataRLPList[9].toUint() // receiptLogIndex
            ];

        RLPReader.RLPItem[] memory logRLPList = logRLP.toList();

        // check child tunnel
        require(childTunnel == RLPReader.toAddress(logRLPList[0]), "RootTunnel:
            INVALID_CHILD_TUNNEL");

        // verify receipt inclusion
        require(
            MerklePatriciaProof.verify(
                inputDataRLPList[6].toBytes(), // receipt
                inputDataRLPList[8].toBytes(), // branchMask
                inputDataRLPList[7].toBytes(), // receiptProof
                bytes32(inputDataRLPList[5].toUint()) // receiptRoot
            ),
            "RootTunnel: INVALID_RECEIPT_PROOF"
        );

        // verify checkpoint inclusion
        _checkBlockMembershipInCheckpoint(
            inputDataRLPList[2].toUint(), // blockNumber
            inputDataRLPList[3].toUint(), // blockTime
            bytes32(inputDataRLPList[4].toUint()), // txRoot
            bytes32(inputDataRLPList[5].toUint()), // receiptRoot
            inputDataRLPList[0].toUint(), // headerNumber
            inputDataRLPList[1].toBytes() // blockProof
        );

        RLPReader.RLPItem[] memory logTopicRLPList = logRLPList[1].toList(); // topics

        require(
            bytes32(logTopicRLPList[0].toUint()) == SEND_MESSAGE_EVENT_SIG, // topic0 is
                 event sig
            "RootTunnel: INVALID_SIGNATURE"
        );
```

```

        // received message data
        bytes memory receivedData = logRLPList[2].toBytes();
        (bytes memory message) = abi.decode(receivedData, (bytes)); // event decodes
            params again, so decoding bytes to get message
        return message;
    }
```

Listing 3.3: `BaseRootTunnel::_validateAndExtractMessage()`

```
    function setPrePurchaseArgs(uint256 _minTokenAmount, uint256 _maxMultiple, uint256
        _limitPerAcc) public onlyExecutor {
        emit PrePurchaseArgsChange(minTokenAmountToPrePurchase,maxPrePurchaseMultiple,
            prePurchaseLimitPerAcc,_minTokenAmount,_maxMultiple,_limitPerAcc);
        minTokenAmountToPrePurchase = _minTokenAmount;
        maxPrePurchaseMultiple = _maxMultiple;
        prePurchaseLimitPerAcc = _limitPerAcc;
    }
```

Listing 3.4: `AngryContract::setPrePurchaseArgs()`

Recommendation Improve the above routines `BaseRootTunnel::_validateAndExtractMessage()` and `RootChainManager::exit()` by validating any public input to filter out unverified ones.

Status The issue has been fixed by this commit: `d3b6e6c`.

3.4 Suggested Versioned Initialization

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Initializable`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize()` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

While examining the upgradeability support as well as the use of current `initializer` modifier, we notice the current implementation can be improved. Specifically, the current `initializer` is provided by the following `Initializable` contract. It supports a rather basic one-time initialization need and does not support any versioned initialization.

```
pragma solidity 0.6.6;

contract Initializable {
    bool inited = false;

    modifier initializer() {
        require(!inited, "already inited");
        _;
        inited = true;
    }
}
```

Listing 3.5: Initializable :: initializer ()

A versioned initialization allows for a new revision number attached to the new upgrade and the upgrade can be performed when the new revision number is larger than the current version one. An example version initialization is shown as follows:

```
  modifier initializer() {
    uint256 revision = getRevision();
    require(
      initializing   isConstructor()   revision > lastInitializedRevision,
      'Contract instance has already been initialized'
    );

    bool isTopLevelCall = !initializing;
    if (isTopLevelCall) {
      initializing = true;
      lastInitializedRevision = revision;
    }

    _;

    if (isTopLevelCall) {
      initializing = false;
    }
```

```
}
```

Listing 3.6: Initializable :: initializer ()

Recommendation Support the versioned initialization to better meet upgrade needs.

Status This issue has been confirmed and the team plans to incorporate the versioned initialization in the future.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

`Matic PoS portal contracts` have a privileged `admin` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., adding new token mappings, registering token predicates, and configuring various system parameters). In the following, we show representative privileged operations in the protocol's core `RootChainManager` contract.

```
    function remapToken(
        address rootToken,
        address childToken,
        bytes32 tokenType
    ) external override only(DEFAULT_ADMIN_ROLE) {
        // cleanup old mapping
        address oldChildToken = rootToChildToken[rootToken];
        address oldRootToken = childToRootToken[childToken];

        if (rootToChildToken[oldRootToken] != address(0)) {
            rootToChildToken[oldRootToken] = address(0);
            tokenToType[oldRootToken] = bytes32(0);
        }

        if (childToRootToken[oldChildToken] != address(0)) {
            childToRootToken[oldChildToken] = address(0);
        }

        _mapToken(rootToken, childToken, tokenType);
    }
```

Listing 3.7: `RootChainManager::remapToken()`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the `admin` is not governed by a `DAO`-like structure. Note that a compromised `admin` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the entire PoS bridge design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and partially mitigated with current 5/7 multisig with representatives from multiple organizations.

3.6 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
```

```
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.8: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `lockTokens()` routine in the `MintableERC20Predicate` contract. If the `USDT` token is supported as `rootToken`, the unsafe version of `IMintableERC20(rootToken).transferFrom(depositor, address(this), amount)` (lines 54 – 58) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
38     /**
39      * @notice Lock ERC20 tokens for deposit, callable only by manager
40      * @param depositor Address who wants to deposit tokens
41      * @param depositReceiver Address (address) who wants to receive tokens on child
           chain
42      * @param rootToken Token which gets deposited
43      * @param depositData ABI encoded amount
44      */
45     function lockTokens(
46         address depositor,
47         address depositReceiver,
48         address rootToken,
49         bytes calldata depositData
50     ) external override only(MANAGER_ROLE) {
51         uint256 amount = abi.decode(depositData, (uint256));
52
53         emit LockedMintableERC20(depositor, depositReceiver, rootToken, amount);
54         IMintableERC20(rootToken).transferFrom(
55             depositor,
56             address(this),
57             amount
```

```
        );
    }
```

Listing 3.9: `MintableERC20Predicate::lockTokens()`

Note that this issue is also present in the `exitTokens()` function within the same contract.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `fb4c154`.

3.7 Single Common Allowance Target For All Predicates

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

`Matic PoS portal contracts` have defined a token predicate interface for all PoS portal predicates. The token predicate interface needs to implement the following two functions: `lockTokens()` and `exitTokens()`. The the names indicate, the first function supports to deposit tokens into PoS portal while the second function allows to withdraw tokens from PoS portal.

The system has so defined a number of token predicates, including `ERC1155Predicate`, `ERC20Predicate`, `ERC721Predicate`, `EtherPredicate`, `MintableERC1155Predicate`, `MintableERC20Predicate`, and `MintableERC721Predicate`. To illustrate, we show below the `ERC20Predicate`. Specifically, we use the `lockTokens()` as our example.

```
    function lockTokens(
        address depositor,
        address depositReceiver,
        address rootToken,
        bytes calldata depositData
    )
        external
        override
        only(MANAGER_ROLE)
    {
        uint256 amount = abi.decode(depositData, (uint256));
        emit LockedERC20(depositor, depositReceiver, rootToken, amount);
        IERC20(rootToken).safeTransferFrom(depositor, address(this), amount);
```

```
    }
```

Listing 3.10: `ERC20Predicate::lockTokens()`

To properly deposit tokens into PoS portal, there is a need to call `approve()` to permit the token predicate to tranfer on behalf of the user. And different predicate requires the user to approve a different address. To make it convenient for users, we suggest to design a single, common allowance target that can then dispatch various `transferFrom()` requests for users.

Recommendation Develop a single, common allowance target, instead of multiple predicates, to simplify the interactions between users and the PoS portal.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Matic PoS portal contracts`. The system presents a unique offering as a bridge that can transfer assets, including include `ERC20`, `ERC721` , `ERC1155` and many other token standards, between the `Polygon` and `Ethereum` when required. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.