PUBLIC

Code Assessment

of the PoS Portal Smart Contracts

April 18, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Polygon with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of PoS Portal according to Scope to support you in forming an opinion on their security risks.

Polygon PoS Portal is a bridge for assets between the RootChain (Ethereum) and the ChildChain (Polygon). Additionally a gas-swapper contract which helps users to aquire MATIC while bridging tokens to Polygon was reviewed.

The most critical subjects covered in our audit are the functional correctness of the bridging mechanism, security of the locked assets and the validation of withdrawals on the RootChain. Security regarding all the aforementioned subjects is high.

The general subjects covered are documentation, efficiency and adherance to the implemented standards. Security regarding all the aforementioned subjects is high. The codebase however could be more consistent: Multiple similar contracts exist where the implementation of the same functionality differs slightly.

This review covered a system already deployed. The actual contracts deployed do not exactly correspond to the version audited, although the changes are mostly of cosmetic nature only. The compiler version + dependencies used are outdated, however no known bug affects the live contracts.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5
• Code Corrected	3
• Specification Changed	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the respective repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

PoS Portal

V	Date	Commit Hash	Note
1	13 February 2023	41d45f7eff5b298941a2547afa0073a6c36b2b9c	Initial Version
2	17 April 2023	ece4e54546a4e075f3a03b2699bc6bd92a5bc065	After Intermediate Report

Gas Swapper

V	Date	Commit Hash	Note
1	13 February 2023	351b8f0097419df1b5174b21bf6685fbd5ca1530	Initial Version

For the solidity smart contracts of the pos portal, the compiler version `0.6.6` was chosen. For the solidity smart contract of the gas swapper, the compiler version `0.8.17` was chosen.

We assume the deployment and initialization of the contracts is done correctly.

2.1.1 Excluded from scope

All files not in the `/contracts` directory of the PoS-Portal repository. Notably system contracts of Polygon, the StateSender, StateReceiver and CheckpointManager(`RootChain.sol`) are not in scope of this review. Files in `/contracts/test` and files used for testing only (`Dummy....sol`) are out of scope.

For the gas swapper the exchange (0x) and mock contracts are out of scope, as well as the plasma bridge which is used to bridge the MATIC tokens.

3 System Overview

Polygon PoS Portal is a bridge for assets between the RootChain (Ethereum) and the ChildChain (Polygon). Supported are assets of the following standards:

- ERC20 Token Standard
- ERC721 Non-Fungible Token Standard
- ERC1155 Multi Token Standard
- Ether

Users can bridge enabled assets as what follows:

From Ethereum to Polygon: Users call the respective deposit functions of the RootChainManager. The asstes will be locked on the RootChain and minted automatically to the receiver on the ChildChain.

From Polygon to Ethereum: Users can claim tokens on the RootChain using the `exit` function of the RootChainManager after having withdrawn the tokens from the child chain. Technically, withdrawing simply requires a transfer event emitted by the ChildToken certifiying the tokens have been burned.

The RootChainManager and the ChildTokens contracts support MetaTransactions: Users can provide signatures for their transaction which can then be executed by any relayer.

3.1 RootChain

3.1.1 RootChainManager

The RootChainManager is the main contract. Communicating messages from Ethereum to Polygon is done using the `StateSender` contract of Polygon. This contract emits an event containing the message to be bridged. Validators of Polygon will pick this event up and add the message to the list of pending state syncs. These messages are automatically executed in sequence on Polygon. The recipient's (`ChildChainManager`) `onStateReceive` function is executed, receiving the data as input. This call has up to 5 000 000 gas available. **If its execution fails for any reason, the message is lost and cannot be re-executed.**

The contract implements role-based access control. There are two privileged roles, `admin` and `mapper`.

Addresses holding the admin role can fully configure the contract using the following functions:

- `setupContractId()`
- `initializeEIP712()`

These functions are intended to be used for initialization only.

- `setStateSender()`
- `setCheckpointManager()`
- `setChildChainManagerAddress()`

These functions allow to update the addresses of the respective contracts.

- `registerPredicate()`

Used to set the address of the predicate contract for a token type. Bridged assets are held at the predicate.

They can access the following very privileged functionality editing the mapping of tokens:

- `cleanMapToken()` Clears a mapping on the RootChain only
- `remapToken()` Remaps a tokens

Furthermore, the admin can manage the roles for addresses.

Addresses with the mapper role can map tokens to enable the movement of this asset via the PoS Portal.

Users can deposit assets using the following functions:

- `depositEtherFor()`
- `depositFor()`

There is a payable fallback function `receive` which invokes `depositEtherFor(__msgSender)`.

The deposit functionality ensures the token is mapped and locks the asset at the predicate contract before bridging the message using the StateSender contract. When the message is received on the ChildChain, the respective assets are automatically minted for the receiver.

After having initiated a withdrawal on the Childchain, a user can exit his tokens on the Rootchain using the RootChainManagers exit function. Messages from the Polygon are read on Ethereum by parsing events emitted by the Child chain. To exit assets, the corresponding transfer event is evaluated and the assets are released from the respective predicate contract to the user.

3.1.2 TokenPredicates

Bridged assets are locked at their respective predicate contract on the RootChain. These predicate contracts must implement functions `lockTockens` and `exitTokens`. Different predicate contract implementations are available depending on the type of token / use case.

- ERC20Predicate: Handles ERC20 tokens.
- ERC721Predicate: Handles ERC721 NFT tokens.
- ERC1155Predicate: Handles ERC1155 tokens.
- EtherPredicate: Handles the locking and release of Ether. `lockTockens()` only tracks the expected amount to be received, the RootChainManager must ensure the actual transfer of Ether.
- MintableERC20/ERC721/ERC115Predicate: `exitTokens()` mints tokens in case of insufficient balance. These predicates hence require miniting rights on the token contract.
- ChainExitERC1155Predicate: Handles ERC1155 emitting the ChainExit() event upon withdrawal from the Child chain.

3.2 ChildChain

3.2.1 ChildChainManager

It is the main contract on the ChildChain implementing `onStateReceive()`, which can only be invoked by the system when bridging message. Two kind of messages are supported, `DEPOSIT` and `MAP_TOKEN`.

- DEPOSIT: completes the deposit initiated on the RootChain by invoking `childTokenContract.deposit()`.
- MAP_TOKEN: completes the mapping of a token initiated on the RootChain by storing the addresses in the respective mappings.

The contract implements role-based access control. There are two privileged roles, `mapper` and `state_syncer_role`.

Addresses with the mapper role can access:

- `cleanMapToken()` Clears a mapping on the ChildChain only
- `mapToken()` Maps a tokens on the ChildChain only

3.2.2 ChildTokens

To represent the assets from the RootChain, token contracts are deployed on the Child chain. Note that the deployment of these tokens must be done manually. Then, to enable a token in the PoS Bridge a mapper must add the token with its corresponding child token to the RootChainManager.

The following token template contracts exist:

- ChildERC20, ERC721, ERC1155
- ChildMintableERC20, ERC721, ERC1155

- MATICWETH (Used to map native Ether of the RootChain)
- UpgradeableChildERC20
- UChildDAI

3.3 Trust Model & Roles

Main contracts such as the Root/ChildChainManager and the TokenPredicates are deployed behind an upgradeable proxy. The owner of the proxy (assumed to be appropriately choosen, e.g. a timelocked or limited multisig) is fully trusted to act honestly and correctly at all time.

The admin role is fully trusted to act honestly and correctly at all time. Misconfigurations may break the system and may result in the loss of assets.

Manager role: fully trusted to map tokens correctly. Incorrect mappings may break the system.

StateSender: Fully trusted system contract.

CheckPointManager: Fully trusted system contract. Source of truth on Ethereum for Polygon blocks.

Polygon Validators: Fully trusted, e.g. not to censor transactions. Must trigger the system call to `StateReceiver` with correct arguments, which triggers `onStateReceive()`.

Users: Untrusted

Tokens: Any external tokens are expected to correctly follow their standards. Tokens are expected not to have non-standard functionalities such as fees on transfer, blacklists, etc. Rebasing tokens specifically are not supported.

3.4 Root and Child Tunnels

Though not used in PoS bridging, `BaseChildTunnel` and `BaseRootTunnel` provide a mechanism for arbitrary message bridging (AMB) to enable bi-directional communication between Ethereum and Polygon. This mechanism basically relies on a loose synchronisation of states between different chains. The aforementioned contracts implement a bridging logic, which can subsequently be inherited by other contracts implementing their own version of `_processMessageFromChild` on the root side and `_processMessageFromRoot` on the child side, in order to process received messages from the other chain.

1. Root to Child chain message passing:

   `RootTunnel` provides the users with a function named `_sendMessageToChild`. This function calls into an already set `stateSender` contract, which publishes the data as an event on the root chain. Later, validators of Polygon fetch this event, and call into `onStateReceive` of `ChildTunnel`, and a custom `_processMessageFromRoot` gets called.

2. Child to Root chain message passing:

   Quite similar to the other direction, `_sendMessageToRoot` on the contract inheriting from `ChildTunnel` gets called, which emits a corresponding event on the child chain. After the block containing this event gets checkpointed on Ethereum, any user can trigger `receiveMessage` to firstly validate the passed-in message in terms of signature, inclusion in the block, and inclusion of the block in the claimed checkpoint. To avoid message reply attacks, the hash of this message gets inserted to a mapping.

These contracts are an early version of the Fx-Portal contracts. It is recommended to use the up to date Fx-Portal contracts instead of these tunnel contracts.

3.5 Trust Model & Roles

RootTunnel: Trustless contract. It has to be initialised with a correct end-point on the child chain and whitelisted in the `stateSender`.

ChildTunnel: Trustless contract. Assumed to be initialised with a correct end-point of the rootchain and be whitelisted in the `stateSender`.

StateSender: Fully trusted.

CheckPointManager: Fully trusted.

Polygon Validators: They are taken as trusted. While due to their authority they can censor transactions.

Admin: Fully trusted.

State Synchroniser: Fully trusted.

Users: Untrusted

3.6 GasSwapper

MATIC is required to pay the transaction fees on Polygon. GasSwapper allows user to swap (Ether for Matic) and bridge (a token): Using 0x Ether will be converted in to MATIC.

The token will be bridged via the PoS bridge (part of this review), while the Matic will be bridged via the Plasma bridge. Any excess Ether (e.g. from the swap) will be refunded to the caller.

3.7 Trust Model & Roles

Trustless contract without privileged roles.

The exchange is expected to behave as expected, the user is fully responsible for the `swapCallData` passed.

The Plasma bridge is expected to work as documented.

Tokens are expected not to have non-standard functionalities such as fees on transfer, blacklists, etc.

4 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5

7.1 ChainExitERC1155Predicate No Exit Event

Design Low Version 1 Specification Changed

No Exit event is defined in `ChainExitERC1155Predicate`. Hence, upon calling `exitTokens` no useful and informative event gets emitted. Furthermore this behavior is inconsistent with the other predicates.

Specification changed:

Polygon has acknowledged lack of an exit event in `ChainExitERC1155Predicate` mentioning that:

> *"Contract is deprecated and was never deployed."*

7.2 ChildChainManager cleanMapToken Emits Wrong Event

Correctness Low Version 1 Code Corrected

By calling `cleanMapToken`, a certain bijection mapping between root and child tokens gets removed. However, the event emitted wrongly indicates a mapping has taken place.

Code corrected:

Polygon defined a new event `TokenUnmapped` which gets emitted once a certain mapping between a root and a child token gets removed.

7.3 Unused ExitedERC721Batch Event

Design Low Version 1 Code Corrected

`ERC721Predicate` defines event `ExitedERC721Batch`, however; `exitTokens` does not support batch exiting of tokens and this event is not used at all.

Code corrected:

Polygon has removed the definition of `ExitedERC721Batch` from their codebase.

7.4 _processMessageFromChild Comment Incorrect

Correctness Low Version 1 Specification Changed

The comment of `_processMessageFromChild()` in `BaseRootTunnel` says that is called from the `onStateReceive` function. This is incorrect. It is actually called from `receiveMessage()`.

Specification changed:

Polygon has corrected the comments on the function `_processMessageFromChild` saying that it is called from `receiveMessage()`.

7.5 `MetaTransactionExecuted` Event Has No Indexed Arguments

Design Low Version 1 Code Corrected

The aforementioned event is defined as

```
event MetaTransactionExecuted(
    address userAddress,
    address payable relayerAddress,
    bytes functionSignature
);
```

None of its arguments are marked as indexed, which could degrade user experience. Indexing fields of events, e.g. addresses, allows to search for them easily.

Code corrected:

Polygon defined `userAddress` and `relayerAddress` as indexed fields of the event `MetaTransactionExecuted`.

```
event MetaTransactionExecuted(
    address indexed userAddress,
    address payable indexed relayerAddress,
```

```
    bytes functionSignature
);
```

7.6 Gas Optimisation Issues Informational

Version 1 **Code Corrected**

The codebase has several inefficiencies in terms of gas costs when deploying and executing smart contracts. Here, we report a list of non-exhaustive possible gas optimizations:

1. `ChildMintableERC1155.deposit` performs a sanity check on `user != address(0)` after decoding `depositData`. This check however has already been done by the RootChainManager.
2. `NativeMetaTransaction.executeMetaTransaction` has a visibility of public. As this function in the current implementation gets called only externally, it ca be defined as external, which subsequently lets memory location of `functionSignature` be calldata. In this way, gas consumption can be reduced.
3. `UpgradableProxy.updateImplementation` checks `_newProxyTo` is non-zero. However, the exact same check is done when calling into `isContract`.
4. `UpgradableProxy.updateAndCall` is a public function. Its visibility can be changed to `external` letting its argument `data` be defined as calldata.
5. `RootChainManager.receive` calls into `_depositEtherFor` with `_msgSender` as the input argument. However, given the fact that sending ETH does not happen through a meta transaction, simply using `msg.sender` can be used.
6. `ITokenPredicate.exitTokens` takes an address as its first argument (`sender`). However, this argument is never used in any implementation of the token predicates.
7. `exitTokens` function for tokens with multiple transfer signatures is implemented as an if-else body, and in each branch same flow of subfield extractions is done. To reduce code footprint, these operations can be moved out of if-else and only logic be kept in each branch.
8. `exitTokens` function in call predicates can have an external visibility and calldata memory location for it `log` argument.
9. In mintable version of each token, inside an if-else statement, it checks whether an excessive amount should be minted and then transfers the actual amount to the receiver. Calling transfer functions can be done outside of if-else to decrease code footprint and reduce deployment cost.
10. `NativeMetaTransaction.getNonce`, which returns current valid nonce of each user. As this view function gets called only externally, its visibility can be changed to external.
11. `ChainExitERC1155Predicate.exitTokens` checks the `withdrawer` is not address zero. However, as the log data fed to it comes from a valid burn event on the child chain, `from` cannot be zero.
12. `BaseChildTunnel.onStateReceive` can be defined as external with `message` having calldata type.
13. `BaseRootTunel.receiveMessage` is never called internally. Therefore, it can be define as external with `inputData` being calldata.

Code corrected:

Polygon has addressed most of the gas optimisation issues. However, for those below they have decided to keep the code as-is:

- 1. *"That is correct but we are in favour of retaining this as an assertion."*
- 5. *"some relayers support ETH metatxs, retaining for backwards compatibility."*
- 7. No further explanations.
- 9. No further explanations.
- 11. *"That is correct but we are in favour of retaining this as an assertion."*

8 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

8.1 Enhance Documenation of Inline Assembly

Informational **Version 1**

Code forked from Biconomy is used to implement support for Meta Transactions. The assembly in function `msgSender()` used to retrieve the sender of the message is not as trivial as it might look. The comment documenting the code section is not appropriately describing what's happening.

```
if (msg.sender == address(this)) {
    bytes memory array = msg.data;
    uint256 index = msg.data.length;
    assembly {
        // Load the 32 bytes word from memory with the address on the lower 20 bytes, and mask those.
        sender := and(
            mload(add(array, index)), //@okaudit-issue todo investigate calculation here, what data do we read?
            0xffffffffffffffffffffffffffffffffffffffff
        )
    }
```

Intuitively the code seems to read 32 bytes past the end of `msg.data`. However, note that for variable length data in memory solidity uses the first 32 bytes to store the length of the data. Hence, `mload(add(array, index))` loads the last 32 bytes of `msg.data` and the code works correctly. Due to the delicate nature of assembly within Solidity, this might be documented appropriately.

9 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

9.1 ChildERC721 Static domainSeparator

Note Version 1

In all variants of ChildERC721, once and only once upon deployment, `domainSeparator` gets calculated using the name of token and chain ID:

```
domainSeperator = keccak256(
    abi.encode(
        EIP712_DOMAIN_TYPEHASH,
        keccak256(bytes(name)),
        keccak256(bytes(ERC712_VERSION)),
        address(this),
        bytes32(getChainId())
    )
);
```

However, in `RootChainManager` and `UChildERC20`, a functionality is devised to let recomputation of `domainSeparator`, e.g. when name of token gets updated. Despite the fact, that forking and a consequent change of chain ID may not be very possible, implementing this functionality in derivations of ERC721Child could make the system more robust.

9.2 Exiting MintableERC721

Note Version 1

MintableERC721Predicate offers several exit possibilities:

- TRANSFER_EVENT_SIG
- WITHDRAW_BATCH_EVENT_SIG
- TRANSFER_WITH_METADATA_EVENT_SIG

Due to the uniqueness of an NFT (`tokenID`) a token can only exist once. However, please consider all withdrawal options emit the Transfer event on the child chain and hence all can be exited using the TRANSFER_EVENT_SIG. This has the following consequences:

For an exit initiated using:

- withdrawBatch: If one transfer has been exited using the TRANSFER_EVENT_SIG, all transfers of the batch must be individually exited using their individual transfer event.
- withdrawWithMetadata: If the TRANSFER_EVENT_SIG is used for the exit, the metadata is lost.

9.3 Minting of ERC721 Tokens

Note Version 1

When using `ChildMintableERC721` and `MintableERC721Predicate`, it is important that only the predicate has minting rights for the token on the root chain.

On the child chain `ChildMintableERC721` allows addresses holding an admin role to mint tokens with arbitrary token ID's given they do not exist on the child chain and have not been withdrawn to the root chain yet.

This protection is only effective when no arbitrary token can be minted on the root chain.

9.4 Recipient of Withdrawn Tokens

Note Version 1

None of the withdraw functions of the child tokens allows to specify the recipient on the root chain. The recipient address is the token owner on the child chain.

It is important to ensure one can access these tokens on the root chain before initiating the withdrawal. Although this generally is not an issue for EOAs, special care must be taken for contracts.

For ERC721/ERC1155 if the recipient is a contract, the contract must implement the appropriate interface or the tokens may be stuck in the bridge as they cannot be exited successfully.