PUBLIC

Code Assessment

# of the Fx Portal Smart Contracts

December 14, 2022

Produced for

polygon

by

CHAINSECURITY

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 Fx Portal according to Scope to support you in forming an opinion on their security risks.

The Fx-Portal allows to seamlessly bridge data between Ethereum and Polygon. Projects can simply build on the provided base contracts and use the provided functions to send/receive messages. Several example implementations are part of the repository, demonstrating the use for a simple state transfer or for bridging tokens.

The most critical aspects covered in our audit are security and functional correctness. For the core part, the mechanism and base contracts of the Fx-Portal, security regarding all the aforementioned aspects is high. The examples, while they showcase the use of the Fx-Portal contracts, lack documentation. Considering that projects may build on top of such example contracts, their functionality / limitations should be properly documented.

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	1
• Specification Changed	1
Medium-Severity Findings	4
• Code Corrected	1
• Specification Changed	1
• Code Partially Corrected	1
• Risk Accepted	1
Low-Severity Findings	8
• Code Corrected	7
• Specification Changed	1

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 `/contracts` directory of the Fx Portal repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	11 August 2022	31dab1ee3da33254a6e6f0c23f3c31a79880e226	Initial Version
2	20 October 2022	dc41712b802a65a0cc2d00ec0833da741dd5ba7c	After Intermediate Report
3	4 November 2022	6ed54a32317afed84d8cb99cfcce8c4fd03b099b	Final Changes

For the solidity smart contracts, the compiler version `0.8.0` was chosen. After the intermediate report the compiler version was updated to `0.8.17`.

2.1.1 Excluded from scope

All files not in the `/contracts` directory of the Fx-Portal repository. Notably system contracts of Polygon, the StateSender, StateReceiver and CheckpointManager(`RootChain.sol`) are not in scope of this review.

2.2 System Overview

Polygon Fx-Portal is a set of smart contracts that enables easy communication between Ethereum and Polygon. It can be thought of as a collection of wrappers for Polygon's `state-sync` mechanism. The repository includes an example use of the Fx-Portal for a generic state-transfer as well as rough examples to implement a bridge for ERC-20, ERC-721 and ERC-1155 tokens.

Contracts `FxRoot` (on Ethereum) and `FxChild` (on Polygon) are used as wrappers for the underlying `state-sync` from Ethereum to Polygon since only whitelisted addresses may call `syncState()` of the `StateSender` contract on Ethereum.

Abstract contracts `FxBaseRootTunnel` (for Ethereum) or `FxBaseChildTunnel` (for Polygon) implement the tunnel logic and can be inherited by smart contracts wishing to use the FxPortal to bridge data.

1. Sending data from Ethereum to Polygon

   `FxBaseRootTunnel` implements `_sendMessageToChild(bytes memory message)` which can be used to initiate a message to be sent to Polygon. From a user/implementor perspective the message is automatically executed on Polygon.

Technically, this function forwards the message and recipient to the external `FxRoot` contract. This `FxRoot` contract encodes and forwards the message to the main `stateSender` contract where an event containing the message and the address of `FxChild` is emitted.

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 (`FxChild`) `onStateReceive` function is executed, receiving the `stateId` and message 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 `FxChild` contract decodes the message and forwards the data to the intended recipient by calling `processMessageFromRoot()`, which `FxBaseChildTunnel` implements. This triggers `_processMessageFromRoot()` which the implementation must implement. Note that the implementation **must** validate the original sender of the message.

2. Sending data from Polygon to Ethereum.

`FxBaseChildTunnel` implements `_sendMessageToRoot(bytes memory message)`. This function simply emits an event `MessageSent` containing the message. In this direction, the message is not executed automatically on the other chain.

After the Polygon block has been checkpointed on Ethereum, anyone may trigger the execution of this message on Ethereum:

`FxBaseRootTunnel` implements `receiveMessage(bytes memory inputData)`. The input must consist of the following:

```
RLP encoded data of the reference tx containing following list of fields
0 - headerNumber - Checkpoint header block number containing the reference tx
1 - blockProof - Proof that the block header (in the child chain) is a leaf in the submitted merkle root
2 - blockNumber - Block number containing the reference tx on child chain
3 - blockTime - Reference tx block time
4 - txRoot - Transactions root of block
5 - receiptRoot - Receipts root of block
6 - receipt - Receipt of the reference transaction
7 - receiptProof - Merkle proof of the reference receipt
8 - branchMask - 32 bits denoting the path of receipt in merkle tree
9 - receiptLogIndex - Log Index to read from the receipt
```

Using this data, it is ascertained that this message was emitted in a valid transaction on Polygon and its consumption is recorded to avoid replaying the same message. `FxBaseRootTunnel` defines `_processMessageFromChild`, which must be implemented by the inheriting contract and process the message.

To use the Fx-Portal to bridge data between Ethereum and Polygon, smart contracts can simply inherit `FxBaseRootTunnel` or `FxBaseChildTunnel` respectively, implement the functions to process received messages, and use the provided functions to send messages.

The following examples are available, which demonstrate how these contracts of the Fx Portal can be used:

- state-transfer
- erc20-transfer
- mintable-erc20-transfer
- erc721-transfer
- erc1155-transfer

The examples have the following functionality:

- state-transfer can be used to send arbitrary data
- erc20-transfer can be used to take an existing ERC-20 token on Ethereum, lock it, and mint a corresponding ERC-20 on Polygon, using a token-template. By burning the Polygon token, the original tokens can be reclaimed.

- erc721-transfer is the same as erc20-transfer, but for ERC-721 tokens.
- erc1155-transfer is the same as erc20-transfer, but for ERC-1155 tokens.
- mintable-erc20-transfer works differently than the other examples. It is used to create a new ERC-20 token on Polygon using a template. This token can be burnt to receive the same amount of a corresponding template ERC-20 token on Ethereum. The Ethereum tokens can be returned to mint the same number of tokens of the original Polygon ERC-20.
- All token examples also allow passing arbitrary data alongside bridging the tokens.

We illustrate an example bridging callpath using the ERC-20 example tunnel:

Bridge ERC-20 from Ethereum to Polygon:

1. User calls `deposit()` on `FxERC20RootTunnel`
2. Tokens get transferred from user to `FxERC20RootTunnel`
3. `FxERC20RootTunnel` calls `_sendMessage()` on `FxBaseRootTunnel`
4. `FxBaseRootTunnel` calls `sendMessageToChild()` on `FxRoot`
5. `FxRoot` calls `syncState()` on `stateSender`
6. `stateSender` emits `StateSynced` event, which Polygon Heimdall validators listen for
7. Heimdall validators pass the event to the Bor layer (EVM chain)
8. On Bor, a special system address `0xffffFFFfFFffffffffffffffFfFFFfffFFFfFFfE` calls `commitState()` on `StateReceiver`.
9. `StateReceiver` checks `msg.sender` and ensures that the messages are sequential and calls `onStateReceive()` on `FxChild`.
10. `FxChild` validates that the `msg.sender` is `StateReceiver` and calls `processMessageFromRoot()` on `FxBaseChildTunnel`
11. `FxBaseChildTunnel` validates `msg.sender` and calls `processMessageFromRoot()` on `FxERC20ChildTunnel`
12. `FxERC20ChildTunnel` mints ERC-20 tokens to the user
13. `FxERC20ChildTunnel` calls `onTokenTransfer()` on user address, ignoring errors

Bridge ERC-20 back from Polygon to Ethereum:

1. User calls `withdraw()` on `FxERC20ChildTunnel`
2. `FxERC20ChildTunnel` burns the user's tokens on Polygon
3. `FxERC20ChildTunnel` calls `_sendMessageToRoot()` on `FxBaseChildTunnel`
4. `FxBaseChildTunnel` emits `MessageSent` event
5. Wait until Polygon validators create a checkpoint including the block with the event on Ethereum
6. User calls `receiveMessage()` on `FxBaseRootTunnel`
7. Validation of the message: `_validateAndExtractMessage()` checks if the exit has already been processed, marks the message as processed, checks that the message comes from the correct ChildTunnel, checks that the receipt is part of the trie. `CheckBlockMembershipInCheckpoint()` ensures the receiptRoot is part of the block, and the block is part of a valid checkpoint using `checkMembership()` with data querried from `CheckpointManager.headerBlocks()` as trusted source.
8. Execution of the message: `FxBaseRootTunnel` calls `_processMessageFromChild()` on `FxERC20RootTunnel`
9. `FxERC20RootTunnel` transfers tokens to user

2.3 Trust Model & Roles

FxRoot: Trustless contract. Assumed to be whitelisted in the `stateSender` contract and initialized with the correct address of `FxChild`.

FxChild: Trustless contract. Assumed to be configured in the `stateSender` contract and initialized with the correct address of `FxRoot`.

FxBaseRootTunnel: Trustless contract. Assumed to be initialized with its corresponding fxChildTunnel and the correct address of `FxRoot`.

FxBaseChildTunnel: Trustless contract. Assumed to be initialized with its corresponding fxRootTunnel and the correct address of `FxChild`.

StateSender: Fully trusted system contract.

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

Polygon Validators: Fully trusted, e.g. to not 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 functionality such as fees on transfer, blacklists, etc. ERC-20 tokens that do not have a return value on transfer are supported. Rebasing tokens specifically are not supported.

3 Security monitoring

Auditing is just one part of a comprehensive smart contract security framework. Next to extensive testing and auditing pre-deployment, security monitoring of live contracts can add an additional layer of security. Contracts can be monitored for suspicious behaviors or system states and trigger alerts to warn about potential ongoing or upcoming exploits.

Consider setting up monitoring of contracts post-deployment. Some examples (non-exhaustive) of common risks worth monitoring are:

1. Assumptions made during protocol design and development.
2. Protocol-specific invariants not addressed/mitigated at the code level.
3. The state of critical variables
4. Known risks that have been identified but are considered acceptable.
5. External contracts, including assets your system supports or relies on, that may change without your knowledge.
6. Downstream and upstream risks - third-party contracts you have direct exposure to (e.g. a third party liquidity pool that gets exploited).
7. Privileged functionality that may be able to change a protocol in a significant way (e.g. upgrade the protocol). This also applies to on-chain governance.
8. Protocols relying on oracles may be exposed to risks associated with oracle manipulation or staleness.

3.1 Project-specific monitoring opportunities

We have identified some areas in Fx Portal that would be well suited for security monitoring.

We classify these into two categories: invariants and suspicious. If an invariant of the system has been broken, there has likely been unexpected behavior. If a suspicious condition is triggered, something has happened that is likely worth investigating.

3.1.1 1. In general for projects using the Fx-Portal

Identified invariant: For every message that is consumed by `receiveMessage()` and `processMessageFromRoot()`, there must be a corresponding event on the other chain.

This invariant could be monitored by listening for events and transactions on both chains and triggering an alert if there is a transaction without a preceding event.

Identified suspicious condition: Calls to `FxChild.onStateReceive()` from the system address should never revert, otherwise messages get irrecoverably lost.

Monitoring for reverts of `onStateReceive()` could detect messages that get lost, for example because of running out of gas.

3.1.2 2. Specific to the examples:

Identified invariant: The `rootToken` balances of the `FxERC20Root/FxERC721/FxERC1155RootTunnel` contract must always be greater or equal than the `totalSupply()` of the `childTokens` on the child chain.

This invariant can be monitored by querying the `rootToken` balance of the `FxERC20RootTunnel` contract and calling `childToken.totalSupply()` in every block, then triggering an alert if `totalSupply()` is smaller.

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	2

- Public Setter Functions Can Be Frontrun Risk Accepted
- Use (Up to Date) Dependencies Code Partially Corrected

Low-Severity Findings	0

6.1 Public Setter Functions Can Be Frontrun

Design Medium Version 1 Risk Accepted

The following functions can only be executed once and have no access controls:

1. `FxBaseRootTunnel.setFxChildTunnel`
2. `FxBaseChildTunnel.setFxRootTunnel`
3. `FxRoot.setFxChild`
4. `FxChild.setFxRoot`
5. `FxERC20.initialize`
6. `FxERC721.initialize`
7. `FxERC1155.initialize`

If deployment and initialization is not done within one transaction it would be possible for a malicious actor to frontrun the deployer's call to the functions and instead call them with malicious values first. This will cause the deployer's function call to revert.

`FxBaseRootTunnel` and `FxBaseChildTunnel` are to be inherited by contracts in order to use the bridging functionality of the Fx Portal. This may lead to problems with their deployment. Implementors should be aware of this behavior, mitigate this and ensure/verify that initialization is done correctly. If their setTunnel functions are frontrun, the contract will need to be redeployed. This can be expensive in terms of gas.

The Wrapper contracts `FxRoot` and `FxChild` for the interaction with the `StateSender` have already been deployed and initialized correctly. If a new instance of one of these contracts is deployed, the deployer must verify that the functions are called correctly. For the Token contracts FxERC20/ERC721/ERC115 used in the examples minimal proxy contracts are deployed the `initialize()` function is called from contracts within the same transactions.

Risk accepted:

Polygon states:

```
It is a known risk that initialization functions can be frontrun, but this
is low-risk since there is no incentive for a malicious actor to do so.
```

6.2 Use (Up to Date) Dependencies

Design Medium Version 1 Code Partially Corrected

Several contracts present in `lib` or `tokens` are copy & pasted from third party repositories. An exception to this pattern is the `SafeERC20` library which is imported from a dependency listed in the `package.json` file. This pattern is generally preferable. Note that several of the copy & pasted dependencies are also from this OpenZeppelin contracts dependency, hence could simply be imported from there.

`Package.json` lists the dependencies and the requirements on the version, while `package-lock.json` allows to fix specific version.

This allows to effortlessly update to newer versions of these contract which may include bug fixes. Note that this must be done with due care as functionality could change. Once a new version has been deemed suitably safe, the new version can be fixed in `package-lock.json`.

Most copy & pasted contracts are old versions, furthermore the version of the OpenZeppelin dependency is outdated. Notably, the implementation of ERC721 contains several changes reloading state after `beforeTokenTransfer()`, which may have updated this data.

Code partially corrected:

The dependencies in `package.json` were changed to more recent versions. `ERC20.sol` and `IERC20.sol` were updated to OpenZeppelin `v4.7.3`.

The other copy & pasted contracts in `lib` have not been updated.

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	1

- mapToken() Callable Only by Mappers Specification Changed

Medium-Severity Findings	2

- Description of toBoolean() Is Incorrect Specification Changed
- FxMintableERC20RootTunnel connectedToken Initialized Incorrectly Code Corrected

Low-Severity Findings	8

- Codehash Variable Type Could Be Set to Immutable Code Corrected
- FxMintableERC20ChildTunnel Has No withdrawTo Function Code Corrected
- FxMintableERC20RootTunnel Events Missing Code Corrected
- Outdated Compiler Version Code Corrected
- Return Value of _checkBlockMembershipInCheckpoint() Code Corrected
- SafeMath Library Is Redundant Code Corrected
- Unused Variable in FxMintableERC20RootTunnel Code Corrected
- _processMessageFromChild Comment Incorrect Specification Changed

7.1 `mapToken()` Callable Only by Mappers

Correctness High Version 1 Specification Changed

In `FxERC20RootTunnel` and `FxERC721RootTunnel`, the `mapToken` function is annotated as follows:

```
/**
 * @notice Map a token to enable its movement via the PoS Portal, callable only by mappers
 * @param rootToken address of token on root chain
 */
function mapToken(address rootToken) public {
```

The function however has no access control, anyone may map a token.

The same function in `FxERC1155RootTunnel` lacks a function description. It also has no access control.

Specification changed:

The comment has been changed to:

```
//@notice Map a token to enable its movement via the PoS Portal, callable by anyone
```

7.2 Description of toBoolean() Is Incorrect

Correctness Medium Version 1 Specification Changed

In RLPReader, the description of toBoolean() states that "any non-zero byte is considered true". The function takes an RLPItem as input.

In RLP encoding, byte values in the range [0x80-0xff] are encoded as 2 bytes like this: [0x81, the_byte]. For RLPItems encoding such values, toBoolean() will revert, since it enforces that the length of the RLPItem is 1. This is a mismatch, as these values are non-zero and should return true according to the comment.

Specification changed:

The comment has been changed to:

```
// any non-zero byte < 128 is considered true
```

7.3 FxMintableERC20RootTunnel connectedToken Initialized Incorrectly

Correctness Medium Version 1 Code Corrected

In the _deployRootToken function of FxMintableERC20RootTunnel, the rootToken's _connectedToken field is initialized as rootToken. This means the rootToken's _connectedToken will be itself, not the childToken on the other chain.

Code corrected:

The _connectedToken is now correctly initialized with the childToken.

7.4 Codehash Variable Type Could Be Set to Immutable

Design Low Version 1 Code Corrected

In the following contracts the variable childTokenTemplateCodeHash could be changed to an immutable:

1. FxERC20RootTunnel.sol
2. FxERC721RootTunnel.sol
3. FxERC1155RootTunnel.sol

This avoids unnessesary and expensive reads from storage, hence reduces the gas consumption.

Code corrected:

These and more variables have been declared `immutable`.

7.5 FxMintableERC20ChildTunnel Has No withdrawTo Function

Design Low Version 1 Code Corrected

The `FxMintableERC20ChildTunnel` has no `withdrawTo` function, unlike the other example contracts. Tokens can only be withdrawn to the same address on the RootChain as the calling address on the ChildChain.

This may make it impossible for some smart contract wallets to bridge tokens, since the user may not be able to deploy the smart contract wallet at the same address on the other chain.

Code corrected:

A `withdrawTo()` function has been added, which takes a `receiver` argument. It calls an internal function `_withdraw()`, which is identical to the previous `withdraw()` function, except that it calls `_sendMessageToRoot()` with the `receiver` address instead of `msg.sender`.

The public `withdraw()` function's functionality is unchanged.

7.6 FxMintableERC20RootTunnel Events Missing

Correctness Low Version 1 Code Corrected

The `FxMintableERC20RootTunnel` contract emits no events when tokens are deposited or withdrawn, which is different behavior than all other example contracts.

Code corrected:

The missing events have been added.

7.7 Outdated Compiler Version

Design Low Version 1 Code Corrected

The project's hardhat config specifies an outdated version of the Solidity compiler.

```
solidity: {
version: "0.8.0",
```

Known bugs in version 0.8.0 are:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1685

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing, the most recent Solidity release is version `0.8.16`.

Code corrected:

The compiler version has been updated to 0.8.17.

7.8 Return Value of _checkBlockMembershipInCheckpoint()

Design Low Version 1 Code Corrected

In FxBaseRootTunnel._validateAndExtractMessage() a call to _checkBlockMembershipInCheckpoint() is made. This internal function has a return value which however is ignored.

Code corrected:

The return value has been removed.

7.9 SafeMath Library Is Redundant

Design Low Version 1 Code Corrected

An old version of the SafeMath library (made for Solidity <0.8) is used in ERC20 and FxERC20MintableRootTunnel.

The main use of SafeMath was previously to revert on arithmetic overflow. As of Solidity 0.8, overflow checks were introduced into the Solidity compiler.

This makes the use of SafeMath redundant.

Code corrected:

The SafeMath library has been removed.

ERC20.sol and IERC20.sol have been updated to OpenZeppelin v4.7.3, which does not use SafeMath. The required IERC20MetaData.sol has also been added.

FxERC20MintableRootTunnel no longer uses SafeMath.

7.10 Unused Variable in FxMintableERC20RootTunnel

Design Low Version 1 Code Corrected

The childTokenTemplateCodeHash variable in FxMintableERC20RootTunnel is declared but never used.

Code corrected:

The unused variable has been removed.

7.11 _processMessageFromChild Comment Incorrect

Correctness Low Version 1 Specification Changed

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

Specification changed:

The comment has been changed to

```
//This is called by receiveMessage function.
```

8 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.

8.1 FxERC20RootTunnel Cannot Map All ERC-20 Tokens

Note Version 1

`mapToken` in `FxERC20RootTunnel` calls `ERC20.decimals()`.

```
//name, symbol and decimals
ERC20 rootTokenContract = ERC20(rootToken);
string memory name = rootTokenContract.name();
string memory symbol = rootTokenContract.symbol();
uint8 decimals = rootTokenContract.decimals();
```

In the ERC-20 standard, `decimals` is optional.

If the `rootToken` does not have a `decimals` function, the call will revert and it will be impossible to map this token.

8.2 _processMessageFromRoot() Must Succeed

Note Version 1

Messages synced from Ethereum to Polygon via the StateSender are executed only once. This execution through `FxChild.onStateReceived()` has 5 million gas available. Should the execution revert for any reason, the message is lost.

The individual implementation of `_processMessageFromRoot()` of smart contracts using the `FxStateChildTunnel` of the Fx Portal must respect that. They should not contain external calls or anything that may revert if lost messages cannot be tolerated.