CRYPTOPIA

Cryptopia.com - Cryptos Token (TOS)

Audit Techspec

Project Overview

The purpose of the Cryptos (TOS) token within Cryptopia, a blockchain game in development, is to facilitate seamless interoperability across multiple blockchains, making it a key component of the broader ecosystem. TOS is an ERC20 asset bridged across multiple blockchains, initially Ethereum and Polygon.

TOS enables a variety of game-centric functionalities, including facilitating economic transactions, governance participation, and enhancing community engagement. Specifically, it allows for the purchase of in-game items, acceleration of construction and crafting processes, and the improvement or repair of items, thereby enriching the gaming experience.

TOS has an initial minting of 10 billion tokens, aligning with the game's tokenomics. The tokens are meticulously locked in vesting contracts by the team, ensuring a strategic distribution that supports the game's long-term economy and growth.

Token Utility documentation:
https://docs.google.com/document/d/1N6p8BFrQja6XpHUIXOPXfKVTODVBRgCkZj2te7v7Bgs/edit

Tokennomics (password: cryptopia2024)
https://blacktokenomics.com/cryptopia/

Cryptopia Game repository:
https://github.com/cryptopia-com/cryptopia-world-contracts

Functional Requirements

1.1. Roles

Default Admin: This is the primary role defined within the Cryptopia token contracts utilized in the ‘**CryptopiaERC20**‘ contract. The ‘**DEFAULT_ADMIN_ROLE**’ is a feature of OpenZeppelin's AccessControl module, which the contract -extends.

The responsibility of the ‘**DEFAULT_ADMIN_ROLE**‘ is to manage the retrieval of external tokens that have been sent to the contract accidentally. This safeguard is implemented to prevent the loss of tokens, ensuring that assets inadvertently transferred to the contract can be returned to their rightful owner. This function is crucial for maintaining the integrity and trust in the Cryptopia token ecosystem, providing a failsafe against accidental token loss.

Depositor (Specific to Polygon): In the ‘**CryptosTokenPolygon**‘ contract, the depositor role, represented by the ‘**ChildChainManager**’ contract at ‘**0xA6FA4fB5f76172d178d61B04b0ecd319C5d1C0aa**‘ on the Polygon mainnet, is needed for the deposit functionality. This role is authorized to call the deposit function, which mints tokens for users on the Polygon network, reflecting deposits made on the root chain (Ethereum). While the depositor role is crucial for depositing tokens, the withdrawal action, which involves burning tokens on the Polygon side to enable their retrieval on the Ethereum network, does not specifically require the depositor role. Instead, the withdrawal process is initiated by users.

User: Participants that engage with the TOS token through transfers, approvals, and bridge transactions for interoperability between Ethereum and Polygon.

Additionally, they utilize TOS for game-centric activities such as purchasing in-game items, speeding up construction and crafting, and item enhancement or repair, directly interacting with the game's economy and governance.

1.2. Features

Transfer of Tokens (User): Users can transfer TOS tokens to any address, facilitating peer-to-peer transactions and interactions within the Cryptopia ecosystem. This basic feature of ERC20 tokens is crucial for the fluid movement of assets within and outside of the game's scope.

Approval Mechanism (User): Users can approve other addresses to spend a specific amount of TOS tokens on their behalf. This feature enables third-party contracts or addresses to interact with TOS tokens securely, such as in-game purchases or automated transactions.

Bridging (User/Depositor): The TOS token supports bridging between Ethereum and Polygon, allowing users to transfer tokens between these blockchains.

Token Retrieval (Default Admin): The '**DEFAULT_ADMIN_ROLE**' has the capability to retrieve tokens that have been accidentally sent to the contract. This feature is a critical safeguard to prevent the permanent loss of tokens, enhancing the security and trust in the TOS/Cryptopia system.

Minting and Burning (Contract-specific actions): The contracts allow for minting new tokens to an address, particularly used during the deposit action on Polygon through the deposit function, and burning tokens from an address, used during the withdrawal action. These mechanisms ensure the token supply can be managed in accordance with bridge operations.

Game-Centric Uses: Beyond ERC20 functionalities, TOS tokens are deeply integrated into the Cryptopia game mechanics. They are used for purchasing in-game items, accelerating construction and crafting times, and enhancing or repairing items, directly impacting the game's economy and user engagement.

1.3. Use Cases

Token Transfer and Approval (User):
Users transfer TOS tokens to other addresses, enabling a dynamic in-game economy and facilitating external transactions.

Users approve third parties to use a certain amount of their TOS tokens, allowing for automated transactions and interactions.

Bridging Tokens Between Ethereum and Polygon (User/Depositor):
Users bridge TOS tokens between Ethereum and Polygon to utilize the game and enhance the token's liquidity across blockchains.

The depositor, identified by the '**ChildChainManager**' contract on Polygon, facilitates the minting of TOS tokens on Polygon following a deposit on Ethereum.

Retrieving Accidentally Sent Tokens (Default Admin):
The default admin retrieves tokens accidentally sent to the contract, safeguarding users' assets and maintaining trust in the Cryptopia ecosystem.

Game-Centric Uses (User):
Users purchase in-game items, such as equipment or resources, using TOS tokens, directly engaging with the game's market.

Users speed up construction, crafting processes, or repair items using TOS tokens, influencing the game's progression and strategy.

For more game-centric uses, view the Token Utility documentation referenced in the project overview.

Technical Requirements

The Cryptos token project is developed using **Solidity**, leveraging the **Hardhat** development environment for comprehensive testing, deployment, and interaction with the smart contracts. **Typescript** is employed for scripting and testing, offering strong typing and advanced language features that enhance the development process.

OpenZeppelin's libraries are utilized for the project, providing secure, audited implementations of standard contracts such as '**ERC20**', which are foundational to the '**CryptopiaERC20**' and related contracts.

https://github.com/OpenZeppelin/openzeppelin-contracts

Structure
The project's structure is organized as follows:

```
├── .deployments
│   └── {network}.contracts.json
├── contracts
│   ├── source
│   │   ├── CryptopiaERC20.sol
│   │   ├── CryptopiaERC20Retriever.sol
│   │   ├── errors
│   │   │   └── AccessErrors.sol
│   │   ├── ethereum
│   │   │   └── CryptosToken.sol
│   │   └── polygon
│   │       └── CryptosTokenPolygon.sol
├── scripts
│   ├── README.md
│   ├── ethereum
│   │   ├── deploy.ts
│   │   └── verify.ts
│   ├── polygon
│   │   ├── deploy.ts
│   │   └── verify.ts
│   └── port.ts
├── test
│   ├── Ethereum.ts
│   └── Polygon.ts
├── .secret (to be added by user)
├── app.ethereum.config.ts
├── app.polygon.config.ts
├── hardhat.config.ts
├── README.md
└── tsconfig.json
```

The '**README.md**' file serves as the entry point for developers, offering essential information about the project's structure, setup instructions, and scripts required for testing and deploying the contracts.

Contracts: The '**./contracts/source**' directory contains the core smart contracts for the project. '**CryptopiaERC20.sol**' is primarily used for the token retrieval functionality, allowing the recovery of tokens sent to the contract by mistake. '**CryptosToken.sol**' handles the minting of an initial 10 billion TOS tokens, establishing the token supply for the ecosystem. '**CryptosTokenPolygon.sol**' adapts the token for use on the Polygon network. Additionally, the '**errors**' subdirectory includes the '**Unauthorized()**' error, which is utilized for access control management.

Scripts: Deployment and verification scripts are located in the '**./scripts**' directory, with separate subdirectories for Ethereum and Polygon, facilitating network-specific operations.

Testing: The '**./test**' directory contains Typescript files for testing the contracts, ensuring their correctness and reliability across Ethereum and Polygon networks.

Configuration: The '**hardhat.config.ts**' file sets up the Hardhat environment, detailing network configurations and plugin integrations for functionalities like Etherscan verification. Additionally, '**app.ethereum.config.ts**' and '**app.polygon.config.ts**' specify Ethereum and Polygon network settings, respectively, which are crucial for the project's deployment and operation on these networks.

Deployments: The '**.deployments**' folder tracks the deployment status of contracts across different networks. It contains JSON files named after each network (e.g., ethereum.contracts.json, polygon.contracts.json), which include details like contract addresses, verification statuses, and deployment transactions.

Secrets: The '**.secret**' file (example provided below) outlines network-specific configurations for managing deployments and interactions with smart contracts.

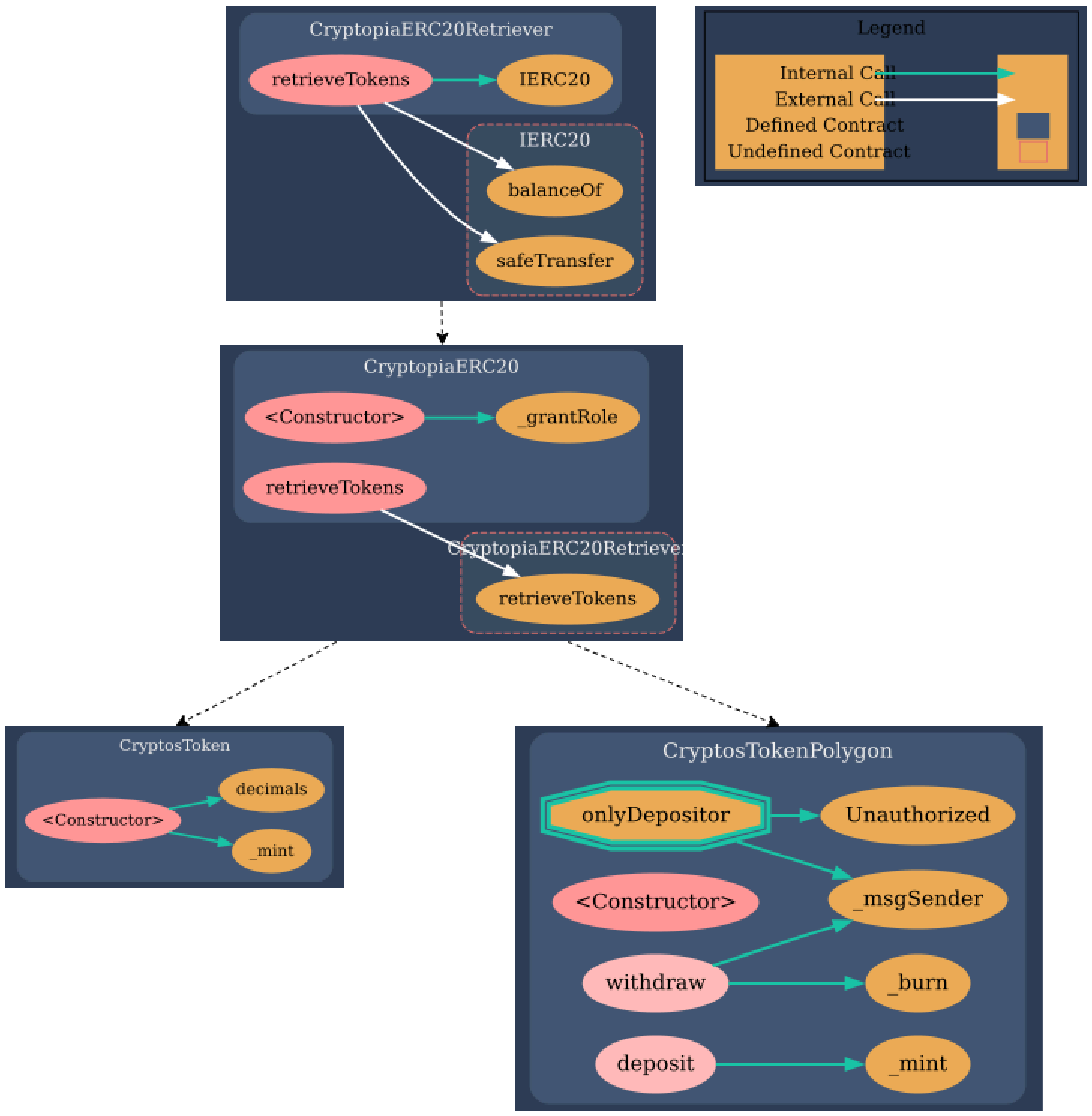
Here's a breakdown of its expected structure and content:

```
{
  "{network}": {
    "mnemonic": "12-word-phrase",
    "etherscan": "api-key"
  }
}
```

Important: The '**.secret**' file contains sensitive information crucial for the security of user's blockchain operations. It should be kept private and not included in the public repository to prevent unauthorized access to project's deployed contracts and funds.

2.1. Architecture Overview

The following chart provides a general view of the contract structures and interactions between different functions.



Created with https://github.com/Consensys/surya

2.3. Contract Information

2.3.1. **CryptosToken.sol**

This contract extends OpenZeppelin's '**ERC20.sol**' via '**CryptopiaERC20.sol**', establishing '**ERC-20**' standard functionalities for the Cryptos (TOS) token. It mints 10 billion TOS tokens at deployment, consistent with the project's tokenomics.

The contract prioritizes simplicity, focusing on core functionalities to minimize the contract's attack surface and enhance security.

2.3.1.1. **Functions**

Constructor: Mints 10 billion TOS tokens to the deployer's address, establishing the total token supply as per the tokenomics.

2.3.2. **CryptosTokenPolygon.sol**

This contract extends OpenZeppelin's '**ERC20.sol**' via '**CryptopiaERC20.sol**', establishing '**ERC-20**' standard functionalities for the Cryptos (TOS) token. It does not mint TOS tokens and has an initial total supply of zero tokens.

The contract serves as the child token on the Polygon network and is a critical component of the integration with the Polygon network, facilitating token transfers via the Polygon bridge.

2.3.2.1. **Assets**

Depositor: Holds the address of the Polygon Bridge's depositor, enabling interaction with the bridge for token deposits.

2.3.2.2. **Errors**

Unauthorized: Utilizes the Unauthorized error from '**../errors/AccessErrors.sol**' for access control, particularly in the context of depositor-only restriction.

2.3.2.3. **Modifiers**

onlyDepositor: Ensures that the token deposit function is callable by the depositor address exclusively, aligning with bridge security protocols.

2.3.2.4. **Functions**

Constructor: Initializes the contract with the depositor's address and sets token details.

deposit(address user, bytes calldata depositData): Called during token deposits on the root chain, minting the specified amount of tokens for the user on Polygon. Restricted to the ‘**depositor’**.

withdraw(uint256 amount): Allows users to burn their tokens on Polygon for withdrawal back to the Ethereum network, facilitating asset transfer across chains.

2.3.3. **CryptopiaERC20.sol**

The ‘**CryptopiaERC20’** contract extends the OpenZeppelin ‘**ERC20**’ and ‘**AccessControl**‘ contracts to implement the ERC-20 standard functionalities for the Cryptos (TOS) tokens. This abstract contract is designed to provide a secure and flexible framework for the token's operations.

2.3.3.1. **Assets**

Access Control: Utilizes OpenZeppelin's ‘**AccessControl**‘ for role-based permissions, granting the ‘**DEFAULT_ADMIN_ROLE**‘ to the contract deployer.

2.3.3.2. **Functions**

Constructor: Sets the token name and symbol while establishing the deployer as the default admin.

retrieveTokens(address tokenContract): A failsafe mechanism allowing the admin to retrieve ERC-20 tokens accidentally sent to the contract. This function ensures that non-Cryptos tokens can be recovered, preventing accidental loss of assets.

2.3.4. **CryptopiaERC20Retriever**

The ‘**CryptopiaERC20Retriever**‘ contract introduces a safeguard mechanism focusing on retrieving ERC-20 tokens that have been accidentally sent to a contract address. It leverages OpenZeppelin's SafeERC20 library for secure token operations.

2.3.4.1. **Assets**

SafeERC20: Ensures the safe transfer of ERC-20 tokens, mitigating risks associated with token retrieval.

2.3.4.2. **Functions**

retrieveTokens(address _tokenContract) virtual: Marked as virtual, this function is designed to be overridden by inheriting contracts. It provides a template for retrieving ERC-20 tokens sent to the contract's address, checking the contract's balance of the specified token, and safely transferring any held tokens back to the caller's address.

The virtual designation indicates that the specific implementation details are intended to be defined in derived contracts, allowing for customized retrieval logic tailored to specific needs or scenarios.

2.4. Third-party

2.4.1. **OpenZeppelin Library**

The OpenZeppelin Library is a foundational framework for secure smart contract development on the Ethereum blockchain. It provides reusable, audited smart contracts for ERC standards, access control, security patterns, and more, significantly enhancing the security and efficiency of blockchain projects.

The Cryptopos (and Cryptopia) project leverages OpenZeppelin's contracts for ERC-20 functionalities, ensuring the Cryptos (TOS) token adheres to industry standards while benefiting from the library's security practices.

Documentation: For detailed guidance on using OpenZeppelin contracts, visit their official documentation.

https://docs.openzeppelin.com/

GitHub Repository: Access the source code and latest updates on OpenZeppelin's GitHub page.

https://github.com/OpenZeppelin/openzeppelin-contracts

2.4.2. **Polygon Bridge**

The Polygon (PoS) Bridge enables asset transfers between Ethereum and Polygon. The bridge operates through a mechanism involving token mapping, deposit, and withdrawal processes, each designed to maintain asset integrity and security across chains.

2.4.2.1. **Bridge Workings and Token Mapping**

Token Mapping: Essential for assets to be eligible for bridging, involving registering a token on Ethereum with its counterpart on Polygon to establish a clear correspondence.

Deposit Process: Involves locking tokens in a contract on Ethereum and minting an equivalent amount on Polygon, facilitated by the ChildChainManager contract.

1. Assets transferred from Ethereum to Polygon are initially locked in a smart contract on the Ethereum network.

2. A set of validators on the Polygon network, operating under a Proof of Stake (PoS) consensus model, verifies the locking of assets on Ethereum.

3. Upon successful verification, the corresponding mint function is executed on Polygon, crediting the user's address with an equivalent amount of tokens.

Withdrawal Process: Tokens are burned on Polygon, and the original assets on Ethereum are unlocked after the burn transaction is validated.

1. For assets returning to the Ethereum network, tokens on the Polygon network are burned.

2. Validators on the Polygon network submit proof of this burn to a smart contract on the Ethereum network.

3. Once the Ethereum network verifies this proof, the previously locked tokens are released to the user's address on Ethereum.

2.4.2.2. **Consensus Mechanism and Validators**

The bridge's security and operational integrity are upheld by a consensus mechanism involving validators within the Polygon network. These validators, operating under a Proof of Stake (PoS) model, are responsible for verifying transactions and ensuring the correct execution of cross-chain transfers.

> **Validators' Role**: Validators play a critical role in the deposit and withdrawal processes by verifying the locking of assets on Ethereum and the burning of tokens on Polygon. Their validation

acts as a secure bridge between the two ecosystems.

Security and Trust: The consensus mechanism relies on the validators' integrity, incentivized by staking MATIC tokens to act in the network's best interest.

2.4.2.3. **Risks and Considerations**

Bridge Security Risks: The bridging process introduces potential vulnerabilities. Users should be aware of these risks, which are mitigated through security measures but not entirely eliminated.

Track Record: The Polygon bridge has a strong security track record. However, the inherent risks of bridging, including smart contract vulnerabilities and reliance on bridge infrastructure security, remain.

Upgradability: The depositor contract is upgradeable, allowing for future enhancements but also introducing uncertainty regarding contract behavior changes.

2.4.2.4. **Simplicity for Security**

The Cryptos (TOS) token contract on Ethereum prioritizes simplicity, focusing on core functionalities without modifications for bridging capabilities. This design minimizes the contract's attack surface, enhancing security. Users opting to keep their tokens on Ethereum can avoid the risks associated with bridging.

2.4.2.5. **Documentation and References**

SDK Documentation: Details on interacting with the bridge via the Polygon SDK.

https://docs.polygon.technology/tools/matic-js/pos/erc20/

Polygon Bridge UI: Official user interface for bridging assets.

https://portal.polygon.technology/bridge

Chainlink Token: Demonstrates adoption of the bridging code by major projects.

https://polygonscan.com/token/0x53e0bca35ec356bd5dddfebbd1fc0fd03fabad39#code

Official GitHub Code: Source code for the ChildToken contract, part of the official Polygon repository.

https://github.com/maticnetwork/pos-portal/blob/master/contracts/child/ChildToken/ChildERC20.sol

Polygon Token Mapper(Proxy): The '**ChildChainManagerProxy**' contract acts as a proxy for the ChildChainManager, facilitating upgradable contract logic without changing the interface address.

https://polygonscan.com/address/0xA6FA4fB5f76172d178d61B04b0ecd319C5d1C0aa

Polygon Token Mapper(Implementation): The '**ChildChainManager**' contract implements the logic for token mapping, deposits, and withdrawals on Polygon, underpinning the bridge's functionality.

https://polygonscan.com/address/0xa40fc0782bee28dd2cf8cb4ac2ecdb05c537f1b5#code