Pantos

Audit of the on-chain component of the Pantos Multi-Blockchain Token System

PANTOS

Sep 24, 2024

Common ⎵ Prefix

Overview

Introduction

Common Prefix was commissioned to perform a security audit on the Pantos Multi-Token System's smart contracts at commit hash cdb08fe6f7f5ad886b7f20a3a975466392cf9924.

The code was of high quality, with detailed comments and an accompanying comprehensive test suite. Along with focusing on security, we ensured that user requests were correctly signed, strictly following the EIP-712 standard. No serious issues were identified.

The files inspected are the following:

```
── access
│   ├── AccessController.sol
│   ├── PantosRBAC.sol
│   └── PantosRoles.sol
├── facets
│   ├── DiamondCutFacet.sol
│   ├── PantosBaseFacet.sol
│   ├── PantosRegistryFacet.sol
│   └── PantosTransferFacet.sol
├── interfaces
│   ├── IBEP20.sol
│   ├── IPantosForwarder.sol
│   ├── IPantosHub.sol
│   ├── IPantosRegistry.sol
│   ├── IPantosToken.sol
│   ├── IPantosTransfer.sol
│   ├── IPantosWrapper.sol
│   └── PantosTypes.sol
├── libraries
│   └── LibAccessControl.sol
├── migrations
│   ├── MigrationTokenBurnablePausable.sol
│   ├── MigrationTokenBurnable.sol
│   ├── MigrationTokenPausable.sol
│   └── MigrationToken.sol
├── BitpandaEcosystemToken.sol
├── PantosBaseToken.sol
├── PantosCoinWrapper.sol
├── PantosForwarder.sol
├── PantosHubProxy.sol
├── PantosHubStorage.sol
├── PantosSimpleToken.sol
├── PantosTokenMigrator.sol
├── PantosToken.sol
├── PantosTokenWrapper.sol
├── PantosWrapper.sol
├── upgradeInitializers
│   └── PantosHubInit.sol
└── wrappers
    ├── PantosAvaxWrapper.sol
    ├── PantosBnbWrapper.sol
    ├── PantosCeloWrapper.sol
    ├── PantosCronosWrapper.sol
    ├── PantosEtherWrapper.sol
    ├── PantosFantomWrapper.sol
    └── PantosMaticWrapper.sol
```

Protocol Description

The protocol enables tokens following the Pantos Token standard to be bridged between EVM compatible chains. As part of this audit, we also reviewed the wrapper contracts, which allow any ERC-20 or EVM-native token to be wrapped into a token adhering to the Pantos standard.

The protocol consists of both off-chain and on-chain components; however, this audit focused solely on the on-chain elements (smart contracts). Users do not interact directly with the smart contracts of the protocol. Instead, they send signed requests (off-chain) to service nodes, specifying the details of the desired action—whether it's a transfer within the same blockchain or a cross-chain transfer. These signed users' requests include details such as the transferred token and amount, the receiver's address, the destination chain (for cross-chain actions), the fee to be paid to the service node, and an execution deadline.

Each service node must be registered in the protocol (in the PantosHub smart contract) and maintain a deposit of PAN tokens (the protocol's native token) as long as it is active. When a node unregisters, it can reclaim its deposit after a specified delay. Service nodes are compensated in PAN for their services and gas costs, with the fee amount defined by the user in their request. Based on the type of request, the service node triggers the appropriate function of the PantosHub smart contract.

The PantosHub smart contract first verifies that the involved blockchains, tokens, and service node are registered within the protocol. The request is then sent to the PantosForwarder contract, which verifies the request's details and signature. For same-chain transfers, the forwarder completes the transfer. For cross-chain token bridging, it burns the tokens on the source chain and assigns a transaction ID to the request. To finalize the cross-chain transfer, the primary validator node must interact with the PantosHub contract on the destination chain and submit a request signed by a quorum of validator nodes, with the required number of signatures set by the protocol owner.

Trust Assumptions

The protocol relies on a few trusted entities to function properly and securely. In this section, we will outline all the trust assumptions made for this audit and briefly discuss the potential consequences if any of these assumptions are not met.

Any entity can register as a service node, provided it holds the required amount of PAN tokens. As such, service nodes are not considered trusted entities. However, trust in service nodes is not necessary because the protocol relies on user-signed requests (the user being the sender of the tokens). While there is no guarantee that service nodes will submit requests to the PantosHub contract in the same order they received them off-chain (and no way to enforce this), users specify a deadline. For same-chain transfers, this deadline marks the latest possible time for the transfer to be executed. For cross-chain transfers, it only indicates the time by which the service node must initiate the bridging process on the source chain. There is no guarantee regarding the finalization of the transfer on the destination chain.

To complete the transfer on the destination chain, a specific number of validator signatures (determined by the protocol owner) is required. The protocol owner, or more specifically, the address assigned the `SUPER_CRITICAL_OPS` role, registers these validators. The protocol remains secure as long as the required number of validator signatures exceeds the number of malicious validators.Validators are expected to finalize actions on the destination chain shortly after they are initiated on the source chain, although this is not enforced by the smart contract code. Furthermore, there is no guarantee that actions will be finalized in the same order they were initiated, or even finalized at all.

There is no control over which tokens are registered and can be bridged by the protocol, as token registration is performed by the token owner rather than by a role within the Pantos protocol. Therefore, users should not inherently trust the registered tokens.They should verify that both the tokens in the original chain and the external token in the destination chain (related to the first) are not malicious. One simple check they should perform is to ensure that, if both tokens have a maximum supply cap, this cap should be the same on both the original and destination chains. If the cap on the destination chain is lower, a cross-chain transaction might be initiated on the source chain but unable to be completed on the destination chain.

Given the fact that the registered tokens are not trusted, the Forwarder uses the `excessivelySafeCall` method, making intentional failures harder to occur and being exploited. Furthermore, service nodes are paid for processing valid requests, regardless of whether the call succeeds or fails. As long as the service node verifies the validity of user requests off-chain, it is protected against DoS attacks related to the smart contracts (though this does not imply protection from off-chain DoS attacks).

We also carefully reviewed the smart contracts concerning the integration of untrusted tokens and found no issues, such as reentrancy vulnerabilities from unusual transfer hooks. But since token registration is permissionless, it should not be seen as a guarantee of the token's safety.

Regarding the Pantos Token standard itself, it’s important to note that each token has a designated forwarder address with the unrestricted ability to mint, burn, and transfer tokens between any two addresses. In the context of the protocol, the forwarder is typically the PantosForwarder contract, which validates and executes user requests. However, a malicious token owner could assign a forwarder address they control for their token. Therefore, users should carefully verify that the forwarder assigned to any token following the Pantos Token standard is a contract with trusted functionality.To address this, we discuss in issue Medium-1 below how the security of this standard could be enhanced.

Finally, trust is also required in the owner of the Diamond Proxy to assign a trusted address as the DEPLOYER. This is important because the DEPLOYER has the authority to call the `diamondCut` function in the DiamondCutFacet, allowing them to add or remove facets of the Diamond, effectively altering its logic.

Disclaimer

Note that this audit does not give any warranties on the bug-free status of the given smart contracts, i.e. the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. Functional correctness should not rely on human inspection but be verified through thorough testing. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

Findings Severity Breakdown

The findings are classified under the following severity categories according to the impact and the likelihood of an attack.

Level	**Description**
Critical	Logical errors or implementation bugs that are easily exploited and may lead to any kind of loss of funds
High	Logical errors or implementation bugs that are likely to be exploited and may have disadvantageous economic impact or contract failure
Medium	Issues that may break the intended contract logic or lead to DoS attacks
Low	Issues harder to exploit (exploitable with low probability), issues that lead to poor contract performance, clumsy logic or seriously error-prone implementation
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable and easier to maintain

Findings

Critical

No critical issues found.

High

No high issues found.

Medium

MEDIUM-1	Lack of user control over forwarder address changes in Pantos Token standard
Contract(s)	PantosBaseToken.sol
Status	**Dismissed**

Description

Tokens following the Pantos Token standard are expected to implement the `IPantosToken` interface. While it is not possible to control how a token implements this interface, it is generally expected that tokens adhering to the standard will use the `PantosBaseToken` contract as a base. As a result, these tokens will have a forwarder address, which is the only address authorized to call the `pantosTransfer`, `pantosTransferFrom`, and `pantosTransferTo` functions, allowing it to transfer tokens between addresses, burn tokens, and mint tokens, respectively.

The forwarder address is set by the token owner. Although the token owner is assigned during the deployment of the token contract and cannot be changed (since the `transferOwnership()` function is overridden to always revert), the `pantosForwarder` address can be updated if the contract implements a function that calls the internal `PantosBaseToken::_setPantosForwarder` function. Ideally, the forwarder should be a trusted contract, likely implementing specific protocol logic, as it does within the Pantos multi-blockchain protocol. However, no standard can entirely prevent malicious actors from adhering to the standard but implementing its functionalities maliciously. At a minimum, the standard should defend against potential errors.

It is expected that a token implementing the Pantos Token standard will have a forwarder contract that enforces protocol-specific logic and a function to change the forwarder in case the protocol requires an update. Even if the update is non-malicious, it may introduce errors. Moreover, even if the change is perfectly safe, users do not provide explicit consent to this change, which could directly impact control over their funds.

Recommendation

We suggest limiting the forwarder's power by requiring user approval for the forwarder address, with this approval being verified in the `pantosTransfer` and `pantosTransferFrom` functions:

```
 function pantosTransfer(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override onlyPantosForwarder {
        require(allowance(spender,_pantosForwarder) > amount, "User hasn't approved
the forwarder");//Common Prefix: check that the user has approved the current
verifier
        _spendAllowance(spender, _pantosForwarder, amount);
        _transfer(sender, recipient, amount);
    }


  function pantosTransferFrom(
        address sender,
        uint256 amount
    ) public virtual override onlyPantosForwarder {
        require(allowance(spender,_pantosForwarder) > amount, "User hasn't approved
the forwarder");
        _spendAllowance(spender, _pantosForwarder, amount);
        _burn(sender, amount);
    }
```

This approval could be set by the users as infinite, meaning users would only need to provide approval for a new forwarder address when it is updated—a rare event that would not significantly complicate the user experience.

However, this solution does not grant users control over the forwarder's minting power, as no simple check can be applied to the `pantosTransferTo` function. Addressing this issue would require protocol-specific solutions. For the Pantos Multi-Chain system in particular, validators could include a signed message from the sender of the `pantosForwarder` address on the destination chain as part of their request.

The current implementation of PantosForwarder verifies that the request is signed by the user/sender.Also, the signed request includes the forwarder address. Therefore the approval check in PantosBaseToken functions is redundant for the current forwarder.However, we believe that incorporating such approval at the level of the token contract would generally enhance the security of the standard, as well as the specific use case within the Pantos multi-chain system, since it allows users to explicitly approve any updates to the forwarder, giving them greater control and protection against e.g. possible bugs in future forwarder contract updates.

Alleviation

Team's comments on the decision to not address the issue:

Pantos clients explicitly sign the address of the Pantos Forwarder contract to be used. Additionally, the Pantos Forwarder contract set in the Pantos Hub is always used. A malicious token can thus not force any other Forwarder contract to be used.Having the Forwarder set in the token contract has the purpose of protecting the token contract itself (against replacing the known Forwarder contract logic with some logic unapproved by the token issuer). Hence, the check in the PantosBaseToken is not redundant, but has no meaning for the sender or protocol, *only for the token contract.Requiring an allowance by the sender for the Forwarder contract would defeat the purpose of enabling users to pay for Pantos transfers using the PAN token. The Pantos Forwarder is meant to be reviewed by any Pantos protocol participants. Hence, senders sign and token issuers set the Forwarder contract to be used.*

Low

LOW-1	Use of the .transfer() method should be avoided
Contract(s)	`PantosCoinWrapper.sol`
Status	**Resolved**

Description

The `unwrap` function in the `PantosCoinWrapper` contract uses the low-level `.transfer()` method to send unwrapped native tokens to the user. The `.transfer()` method forwards a fixed amount of gas (2300), which is enough to log an event but insufficient for executing more complex logic. Historically, it was recommended as a defense against reentrancy, as it prevents complex actions, such as reentrant calls to the original contract, from being executed in the receiver's fallback function. However, this assumption only holds if gas costs remain constant, which is not the case. Since gas costs are subject to change, smart contracts should not rely on any specific gas limitations. By using `.transfer()`, you create an undesirable dependency on gas costs.

This also complicates integration with other protocols. If the receiver is a contract that implements additional logic in its fallback function—such as accounting for received native tokens—the gas limitation imposed by `.transfer()` may not be sufficient, causing the transaction to fail.

Furthermore, this contract follows the checks-effects-interactions pattern (first burning the wrapped tokens, then sending the deposited native tokens to the user), which mitigates the risk of reentrancy.

Recommendation

We recommend using `.call()` instead of `.transfer()` and verifying that the call was successful:

```
function unwrap(uint256 amount) public override whenNotPaused onlyNative {
        _burn(msg.sender, amount);
        (bool success, ) = payable(msg.sender).call.value(amount)("");
        require(success, "Transfer failed.");
    }
```

Alleviation

The team fixed the issue at commit hash [cdb0340c6e887d6f9cb23178f231f029718fba7c](), implementing our suggestion to use `call` instead of `transfer`.

LOW-2	The protocol can extend indefinitely the unbonding period of the service nodes' deposits
Contract(s)	`PantosRegistryFacet.sol`
Status	**Resolved**

Description

Service nodes can unregister from the protocol and withdraw their deposit in two steps. First, they call `unregisterServiceNode`, which deactivates the service node, preventing it from submitting requests to the Hub contract. This also sets the `serviceNodeRecord.unregisterTime` to the current block timestamp. Later, the service node can call `withdrawServiceNodeDeposit` to retrieve its deposit, but only after a future time `t` that is greater than `serviceNodeRecord.unregisterTime + s.unbondingPeriodServiceNodeDeposit.currentValue`.

However, the value of `s.unbondingPeriodServiceNodeDeposit.currentValue` can change between the time of unregistration and the withdrawal request. For instance, if the `parameterUpdateDelay` at the time of unregistration is shorter than the unbonding period, the owner (specifically, the MEDIUM_CRITICAL_OPS address) could call both `initiateUnbondingPeriodServiceNodeDepositUpdate` and `executeUnbondingPeriodServiceNodeDepositUpdate`, potentially extending the unbonding period before the service node can withdraw its deposit. Repeating this process could extend the unbonding period indefinitely. Additionally, the owner could shorten the delay time and apply multiple updates to further manipulate the unbonding period.

Recommendation

While the protocol owner and all the addresses granted special roles are considered trusted, it would be advisable to limit this power by locking in the unbonding period at the time of the service node's unregistration. Furthermore, enforcing a minimum value for the delay parameter and setting maximum bounds on the unbonding period would provide additional safeguards against potential abuse.

Alleviation

The team fixed the issue at commit hash The team addressed the issue in commit cdb0340c6e887d6f9cb23178f231f029718fba7c by locking the unbonding duration during the service node unregistration process.

LOW-3	Compatibility issues with non-standard ERC-20 Tokens in wrap()
Contract(s)	`PantosTokenWrapper.sol`
Status	**Resolved**

Description

In the `wrap` function, a check is performed to confirm whether the user's tokens were successfully transferred to the contract. If successful, the function mints wrapped tokens to the user; otherwise, it reverts. This is achieved using the following `require` statement:

```
require(
        IERC20(_wrappedToken).transferFrom(
                msg.sender,
                address(this),
                amount
            ),
            "PantosTokenWrapper: transfer of tokens failed"
        );
```

Observe that the current implementation assumes that the token follows the standard ERC-20 specification, which returns a boolean on transfers. This can lead to issues when interacting with non-standard tokens that either do not return a value or return types other than boolean.

Recommendation

To ensure compatibility with both standard and non-standard ERC-20 tokens, we recommend using the SafeERC20 library for token transfers.

Alleviation

The team fixed the issue at commit hash cdb0340c6e887d6f9cb23178f231f029718fba7c by using the `SafeERC20` library for ERC20 transfers.

Informational/Suggestions

INFO-1	Missing check in _verifyTransferTo that source and destination blockchain IDs are different
Contract(s)	`PantosTransferFacet.sol`
Status	**Dismissed**

Description

The `_verifyTransferTo` function should revert if the source `blockchainId` in the request matches the Id of the current blockchain. While this scenario is unlikely to occur with honest validators—since the `transferFrom` function (which initializes the bridging process) already verifies that the source and destination blockchains are different—adding this simple check in the contract would provide an additional layer of security.

Alleviation

The team has decided not to address this issue, as adding this extra check would block reverse transfers.

INFO-2	Modifiers applied twice
Contract(s)	`BipandaEcosystemToken.sol, MigrationToken.sol, MigrationTokenBurnable.sol, MigrationTokenBurnablePausable.sol, MigrationTokenPausable.sol, PantosForwarder.sol, PantosToken.sol, PantosWrapper.sol`

Status	**Partially resolved**

Description

In the `setPantosForwarder` function, the `onlyOwner` modifier is applied, but this function also calls the `_setPantosForwarder` function, which redundantly applies the same modifier. Similarly, the `pause` and `unpause` functions use the `whenNotPaused` and `whenPaused` modifiers, respectively. However, these functions call the internal `_pause` and `_unpause` functions from the OpenZeppelin `Pausable` contract, which already apply the same checks, making the modifiers redundant.

Also, the `onlyPantosHub` modifier in `PantosForwarder::verifyAndForwardTransfer`, `verifyAndForwardTransferFrom` and `verifyAndForwardTransferTo` is redundant since these functions call `verifyTransfer`, `verifyTransferFrom` and `verifyTransferTo` respectively where the `onlyPantosHub` modifier is also applied.

Alleviation

The team fixed the issue of the `onlyPantosHub` modifier applied twice in the `PantosForwarder` contract at commit hash cdb0340c6e887d6f9cb23178f231f029718fba7c, but not the other instances of the issue.

INFO-3	Inconsistent role usage in the setPantosToken function
Contract(s)	`PantosRegistryFacet.sol, IPantosRegistry.sol`
Status	**Resolved**

Description

The comment above the `IPantosRegistry::setPantosToken` function states that this function should only be callable by the `DEPLOYER` role and only when the protocol is paused. However, in the actual implementation within the `PantosRegistryFacet`, the `onlyRole(PantosRoles.SUPER_CRITICAL_OPS)` modifier is applied (likely because the function calls `register`, which can only be executed by the `SUPER_CRITICAL_OPS` role when the protocol is paused).

To avoid confusion and ensure transparency with users, it is important to clarify which specific operations are permitted for each role and how they interact with the protocol's paused state.

Alleviation

The team clarified, fixing the comment, that only the super critical ops role should have permission to call the `setPantosToken` function.

INFO-4	Missing forwarder validation in registerToken function
Contract(s)	`PantosRegistryFacet.sol`
Status	**Resolved**

Description

In the `registerToken` function, there should be a check to ensure that the forwarder of the token (`IPantosToken(token).getPantosForwarder()`) matches the protocol's forwarder (`s.pantosForwarder`). Otherwise, a token could be registered in the Hub contract but would not be usable within the protocol, because the protocol forwarder will not be able to call the `pantosTransfer`, `pantosTransferFrom`, and `pantosTransferTo` functions, since it would not be the same as the token forwarder.

Alleviation

The team fixed the issue at commit hash cdb0340c6e887d6f9cb23178f231f029718fba7c by implementing the suggested check.

INFO-5	Unused internal function
Contract(s)	`PantosBaseToken.sol`
Status	**Dismissed**

Description

The `PantosBaseToken::_unsetPantosForwarder` function is marked as internal and is not called by any other function in the PantosBaseToken or any other contract inheriting it. Its only use is within the testing environment. Therefore, it would be better to remove this function from the final codebase to reduce unnecessary complexity and maintain clean, production-ready code.

Alleviation

The team dismissed this issue, providing the following explanation:

The `PantosBaseToken` contract is meant to be used by external token issuers as well. They might want to unset the Forwarder contract address. Hence, the `_unsetPantosForwarder` method should be kept as it is right now.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of scientists and engineers who specialize in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.