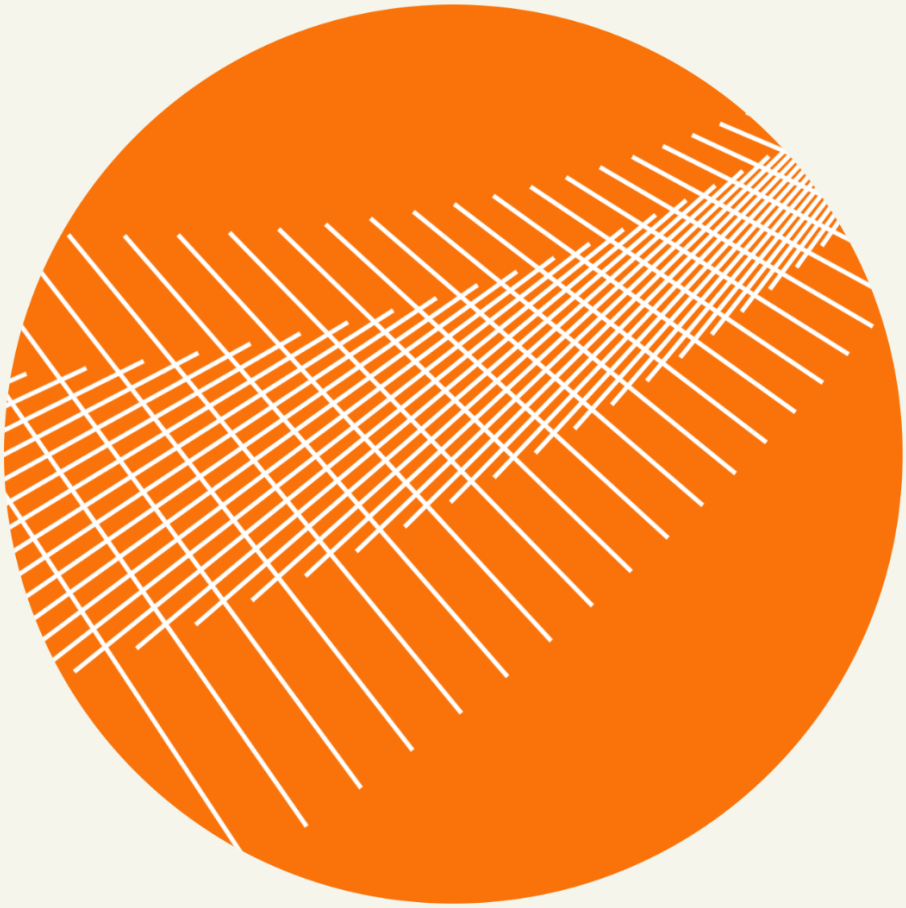
CTDSEC

YOU ARE PROTECTED



TERMS

Smart contract security audit

PWR - Auction Crosschain

Table of Contents

1.0 Introduction

1.1 Project engagement

During September of 2025, PWR team engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. PWR provided CTDSec with access to their code repository and whitepaper.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the network's fast-paced and rapidly changing environment, we at CTDSec recommend that PWR team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the PWR contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Source file:

pwrAuctionCrossChain.zip:

876d0adb5c0e271ea3e24059eaa9604524bf156bb3e9f5879511d04ffe9d75f7

Fixed version [pwrAuctionCrossChainv2.zip] :

1dd5aac1d2a779c56ff714cca34589b32bbf3ee934c1bc188daf230cbcad31f3

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.
1	Compiler warnings.
2	Race conditions and Reentrancy. Cross-function race conditions.
3	Possible delays in data delivery.
4	Oracle calls.
5	Front running.
6	Timestamp dependence.
7	Integer Overflow and Underflow.
8	DoS with Revert.
9	DoS with block gas limit.
10	Methods execution permissions.
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.
12	The impact of the exchange rate on the logic.
13	Private user data leaks.
14	Malicious Event log.
15	Scoping and Declarations.
16	Uninitialized storage pointers.
17	Arithmetic accuracy.
18	Design Logic.

19	Cross-function race conditions.
20	Safe Zeppelin module.
21	Fallback function security.
22	Overpowered functions / Owner privileges

3.0 Security Issues - Smart contracts

3.1 High severity issues [4]

1. Cross-chain vesting flow is incompatible (remote claims revert) [Fixed ✅]

Contract: TokenVesting.sol, BidCollector.sol, PWR Token.sol

Function: TokenVesting.receiveApproval

BidCollector.claimTokens, _tokenTransfer

PWRtoken.approveAndCall

Problem: On destination chain, BidCollector._tokenTransfer calls approveAndCall(tokenVesting, vestedAmount, data). This triggers TokenVesting.receiveApproval which enforces from == auctionAddress. Here, from equals BidCollector (not Auction), so NotFromAuction() always reverts. Result: cross-chain claims (20% instant + 80% vest) cannot be completed so basically users are DoS'd.

Recommendation: Adopt one consistent design:

Whitelist remote distributor: add authorizedDistributors[BidCollector] = true and accept from == auctionAddress || authorizedDistributors[from] in receiveApproval.

Or keep vesting only on the Auction chain, dont vest on destination, only send the 20% cross-chain.

Or compose back to Auction for vesting: perform approveAndCall from Auction after a remote claim request.

Fix: okenVesting.receiveApproval now requires:

```
msg.sender == tokenAddress, and
from == PWRholderAddress
```

Plus there is a new owner-only setter setTokenHolderAddress(address _holder) so you can assign the on-destination holder (for example the BidCollector). When BidCollector calls PWRtoken.approveAndCall, the from will be BidCollector, which will match the configured PWRholderAddress, so it no longer reverts.

2. Wrong buyer identity recorded in cross-chain path [Fixed ✅]

Contract: Auction.sol

Function: _lzReceive → buyInAuctionCrossChain

Problem: Cross-chain buyer metadata is set using msg.sender (the endpoint) instead of payload fields. Code sets:

```
info.buyerAddress = msg.sender;
info.spender      = msg.sender;
```

This corrupts buyerInfoMap for that userkey, breaking identity-based lookups and any vesting/accounting relying on the stored spender.

Recommendation: Use payload values:

```
info.buyerAddress = p.buyerAddress;
info.spender      = p.spender;
```

Keep _eid from the origin.

Fix: In Auction._lzReceive to buyInAuctionCrossChain, buyerInfoMap is now populated with payload values, not msg.sender:

```
info.buyerAddress = p.buyerAddress;
info.spender      = p.spender;
info.eid          = _eid;
```

3. Owner can drain user escrow at any time [Centralization] [Fixed ✅]

Contract: BidCollector.sol, Auction.sol

Function: BidCollector.drain

Auction.drain, Auction.drainRemainTokens

Problem: Admin can transfer all escrowed USDT/USDC (and remaining PWR) to treasury without time or state restrictions. A compromised key or malicious admin can sweep funds mid-sale.

Recommendation: Gate drains behind stronger controls:

Require halted == true AND sale ended, plus a timelock.

Use a multisig for ownership.

Consider a 2-step emergency: announce - delay - execute.

Fix: Drain now requires a two-step process:

requestDrainEscrow() : onlyOwner, whenHalted and whenEnded, stores a timestamp.

drain() : onlyOwner after DRAIN_DELAY = 24h from the request.

Same pattern for remaining token drains in Auction. Funds are transferred using SafeERC20.

4. Claim path reverts atomically, users remain perma-blocked [Fixed ✅]

Contract: BidCollector.sol

Function: claimTokens, _tokenTransfer

Problem: claimTokens sets isClaimed = true then performs the instant transfer and vesting call. Because vesting currently reverts (see 1.), the whole tx reverts and isClaimed rolls back, users cannot succeed nor make partial progress.

Recommendation: After fixing the vesting incompatibility, harden sequencing:

Transfer 20% first: if vesting fails, record the remaining amount for a retry (don't revert the entire flow).

Alternatively, pre-check vesting preconditions and revert before changing any state or moving funds.

Fix: claimTokens now:

```
Marks isClaimed = true.
Transfers the 20% instant via safeTransfer immediately.
Attempts vesting with approveAndCall in a try/catch.
```

If vesting fails, it does not revert, now it stores tokenToClaim = vest for a later call to retryVesting(userkey) which re-attempts approveAndCall.

3.2 Medium severity issues [2]

1. Silent ERC20 transfer failures on drains [Fixed ✅]

Contract: Auction.sol

Function: drain, drainRemainTokens

Problem: These functions call transfer and ignore boolean return values. Non-standard tokens (for example some USDT variants) may return false without reverting, causing silent failure and inconsistent state.

Recommendation: Check return values and revert on failure.

Fix: Drains use SafeERC20.safeTransfer, which reverts on failure (no silent false).

2. Risky ordering in submitBid (state & funds after cross-chain send) [Fixed ✅]

Contract: BidCollector.sol

Function: submitBid

Problem: Function sends the LZ message before pulling funds and before updating buyerInfo. While atomicity saves you on revert, this ordering expands the surface for partial changes if refactored or if external hooks evolve.

Recommendation: Reorder: validate - pull funds via transferFrom - update buyer info - _lzSend - emit.

Fix: Ordering is now: validate then transferFrom funds then record buyer info and finally send LZ message. Funds are pulled before messaging and state is updated before send.

3.3 Low severity issues [1]

1. Additive allowance in approveAndCall could surprise users [Acknowledge ✅]

Contract: PWR Token.sol

Function: approveAndCall

Problem: The function sets allowance to allowance + amount (additive). This may confuse integrators expecting 'set to exactly amount', and it keeps legacy ERC-20 allowance pitfalls alive (though mitigated by immediate call).

Recommendation: Prefer setting allowance to exactly amount for the atomic pattern, or clearly document the additive behavior.

3.4 Informational Findings [1]

1. Mutable pricing/issuance knobs during active sale (governance risk) [Acknowledge ✅]

Contract: Auction.sol

Function: setUSDWEI (while active), setFinalMinPrice, setUnitConversion, setReferralBonus, setHalted

Problem: Admin can change core economic parameters mid-sale, altering price and allocations. Not a bug, but a trust/governance risk.

Recommendation: Add a timelock and public announcements for parameter changes; optionally freeze critical params after an initial window.

4.0 Summary of the audit [PASS ✅]

Several high-medium severity gaps can disrupt user experience or disable features. With the recommended mitigations, the contract can achieve a robust security posture fit for production deployment. The audit result is **failed** until further review by the development team.

UPDATE: Following the development team's review, all critical vulnerabilities have been mitigated/reviewed, and the audit result is satisfactory.

Vulnerability Level	Total	Pending	Not Apply	Acknowledged	Resolved
High	4				4
Medium	2				2
Low	1			1	
Informational	1			1	