SECURITY AUDIT REPORT: PRIZEDISTRIBUTOR PROTOCOL

PROJECT	PrizeDistributor Protocol
REPOSITORY	https://github.com/molalign8468/30-day-security-sprint
COMMIT HASH	2a112ec28ab16295810e52aa4c410ef6769820a4
AUDIT DATE	February 24, 2026
AUDITOR	Molalign Getahun

Executive Summary

Status: ***CRITICAL RISK***

The PrizeDistributor contract contains critical-severity vulnerability due to Denial of Service via Unexpected Revert. The flaw stems from a "Push-Payment" architecture, where the contract attempts to send Ether to multiple recipients in a single transaction. This allows a single malicious actor to intentionally block the transaction, permanently locking all contract funds.

This flaw leads to a permanent deadlock of the protocol's assets. Because the distributed flag is only set to true at the end of the loop, a single malicious winner can indefinitely block the distribution, rendering the withdrawLeftover() function inaccessible and locking the remaining balance in the contract. The contract is unsafe for mainnet deployment in its current state. Immediate remediation is required before any production use.

Severity	Count	Status
Critical	1	Unresolved
High	0	-
Medium	0	-
Low	0	-
Informational	0	-

Scope & Methodology

Scope

File	Type	Logic
Day5_DoS.sol	Smart Contract	deposit,setWinners, distribute and withdrawLeftover functions logic

Methodology:

- ✓ Manual code review
- ✓ Threat modeling
- ✓ Common vulnerability pattern analysis (SWC)

Severity Rating

Severity	Description
Critical	Full fund loss or protocol takeover
High	Significant fund risk
Medium	Logic or trust issues
Low	Minor risk
Informational	Best practices

Findings Summary

ID	Severity	Title
CRIT-01	Critical	Denial of Service (DoS) via Unexpected Revert

Detailed Findings

[CRIT-01] Denial of Service (DoS) via Unexpected Revert

Severity: Critical

Location: Day5_DoS.sol:L28–L38

Reference: SWC-113 – DoS with Failed call

Description

The distribute() function implements a "Push-Payment" strategy that is inherently vulnerable to a Denial of Service. By using require(success, "transfer failed"); inside a for loop, the contract ensures that if any single recipient (a "winner") cannot or will not accept the Ether transfer, the entire execution reverts.

A malicious winner can deploy a contract with a reverting receive() function. Because the distributed state variable is only toggled at the very end of the loop, this single failure prevents the entire batch from being processed, effectively holding all contract funds hostage.

Impact

- ✓ Funds remain locked in contract
- ✓ Legitimate winners cannot receive compensation
- ✓ Economic assumptions broken
- ✓ The contract enters an unrecoverable state called Permanent Deadlock

Risk Evaluation

- ✓ **Impact**: Results in a permanent deadlock of the contract state. All funds remain locked indefinitely.
- ✓ **Likelihood**: Any winner can easily break the entire protocol by using a contract that rejects Ether, causing a mandatory revert that triggers a permanent deadlock.
- ✓ **Overall Severity**: Critical.

Proof of Concept

Vulnerable Contract

```
contract PrizeDistributor  {
    address public owner;
    mapping(address => uint) public balances;
    address[] public winners;
    bool public distributed;

    modifier  onlyOwner {
        require(msg.sender == owner,"not owner");
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function setWinners(address[] calldata _winners) external  onlyOwner {
        require(!distributed, "already distributed");
        winners = _winners;
    }

    function distribute() external onlyOwner{
        require(!distributed, "already distributed");
        require(winners.length > 0, "no winners  set");

        uint amountPerWinner = address(this).balance / winners.length;

        for (uint i = 0; i < winners.length; i++) {
            address winner = winners[i];
            (bool success, ) = winner.call{value:  amountPerWinner}("");
            require(success, "transfer failed");
        }

        distributed = true;
    }
    function withdrawLeftover()  external  onlyOwner{
        require(distributed, "not distributed yet");
        (bool ok,) = payable(owner).call{value:address(this).balance}("");
        require(ok,"Withdraw  fail");
    }


    receive() external payable {}
}
```

Attacker Contract

```
contract MaliciousReceiver   {
    receive() external payable{
        revert("I refuse to accept money");
    }
}
```

Recommendations

- ✓ Replace the current "Push-Payment" logic with a Pull-Payment pattern. Instead of the contract sending Ether to everyone at once, store each winner's share in a mapping and allow them to withdraw it individually. This isolates the failure of a single recipient and prevents a permanent deadlock.

```
mapping(address => uint) public pendingWithdrawals;

function distribute() external  onlyOwner {
    uint amountPerWinner = address(this).balance / winners.length;
    for (uint i = 0; i < winners.length; i++)  {
        pendingWithdrawals[winners[i]]  += amountPerWinner;
    }
    distributed = true;
}

function withdraw() external  {
    uint amount = pendingWithdrawals[msg.sender];
    pendingWithdrawals[msg.sender] = 0;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "withdraw failed");
}
```

Conclusion

Deployment should not proceed until the issue is fixed and the contract is re-audited.