securing.pl

Reentrancy attack in smart contracts - is it still a problem? - Securing

Paweł Kuryłowicz Principal IT Security Consultant Hackflix Product Lead

8-10 minutos

Reentrancy is as old as Solidity itself, and even older because it's not the only programming language it appears in. It got a lot of attention after one of the hacks that took place in 2016 where millions of dollars were stolen. It's been over 5 years, has something changed in this topic? Well, not much. In this article I have touched major omissions about security in this topic.

TL;DR

- Always use the Checks Effects Interactions pattern.
- Use mutex on any function that you are unsure of.
- Watch out for cross-function and cross-contracts unsafe external calls.
- Verified and secure components != verified and secure system.

- …and always use the Checks Effects Interactions pattern.

Reentrancy – vulnerability or not?

Here comes the favorite engineer's answer – it depends. Reentrant behavior might not be a security issue. **The vulnerability occurs when the behavior is unexpected and could be used to the detriment of the project.** What's more, reentrancy is rather one step of the exploitation, not the vulnerability itself.

Knowing that, we can finally explain when it occurs. One of the shortest definitions of reentrancy (or rather unsafe external call) is when contract A calls contract B, and contract B calls contract A, when A still has not updated its state and it leads to some unexpected harmful behavior.

Okay, but is this reentrancy attack a significant problem?

Noting that 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 mln, because of a fake token contract and a reentrancy exploit.
- The SURGEBNB hack (August 2021) – $4 mln, seems to be a reentrancy-based price manipulation attack.
- CREAM FINANCE hack (August 2021) – $18.8 mln,

reentrancy vulnerability allowed the exploiter for the second borrow.

- Siren protocol hack (September 2021) – $3.5 mln, AMM pools were exploited through reentrancy attack…
- … and more.

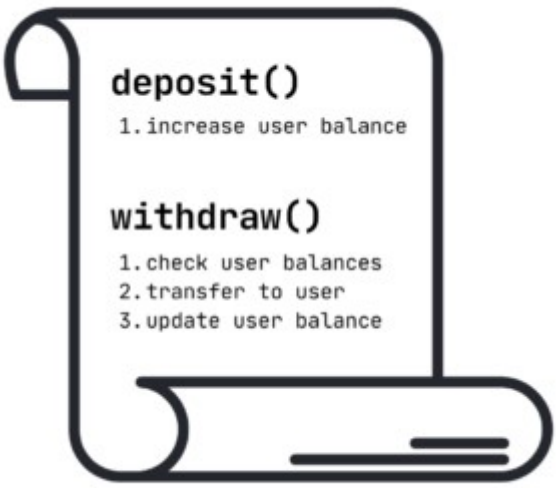
Yes, it is a significant problem.

An abstract example of reentrancy attack

Okay, but how to spot reentrancy in the code? Look at all functions that have some external calls and decide if those are secure or not based on their logic. Let's take a look at high-level example below:

High-level example of the vulnerability:

Let's imagine that a vulnerable contract has only two functions:
1. deposit() – that increases user balance,
2. withdraw() – that checks user balance, makes transfers, and updates user balance.



Why is this logic not good? Because of an external call which is made during the transfer, malicious attackers can prepare a contract that will call the withdraw function again before updating their balance.

Attack scenario:

1. Vulnerable smart contract have 10 eth.
2. An attacker uses a deposit function to store 1 eth.
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.
- (attacker balance has NOT been updated yet)
- Fallback function on received eth calls *withdraw* function again.
5. 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 and again until the attacker will drain all the funds stored on the contract.

What would it look like in actual code?

Vulnerable contract:

```
contract VulnerableBank {

    mapping (address=>uint256) balance;

    function deposit () external payable {
        balance[msg.sender]+=msg.value;
    }
    function withdraw () external payable{
        require(balance[msg.sender]>=0,'Not
enough ether');

payable(msg.sender).call{value:balance[msg.sender]}
("");
        balance[msg.sender]=0;
    }
    function banksBalance () public view
returns (uint256){
        return address(this).balance;
    }
    function userBalance (address _address)
public view returns (uint256){
        return balance[_address];
    }
}
```

Imagine a bank contract that has 10 eth. The contract includes the eth of all users and allows them to withdraw the amount they deposit whenever they wish. But can you withdraw someone’s eth?

Sure, you can even empty the entire contract!

Here, the withdraw function is vulnerable. There is no mutex and the contract is not designed according to the CEI (Checks Effects Interactions) pattern. In this particular case the interaction is before the effect, and to be more precise, token transfer is before balance update.

So how will the attack scenario look like here in detail?

1. The attacker creates a malicious contract
2. Malicious contract: Calls the deposit function on a vulnerable contract to increase its balance by 1 eth.
3. Vulnerable contract: It records the transfer and increases the attacker's contract balance.
4. Malicious contract: Calls the withdraw function on a vulnerable contract to extract everything they deposited.
5. Vulnerable contract: Checks if the stored balance of the attacker is greater than or equal to 0.
6. Vulnerable contract: Yes, it is greater than 0 due to the transfer in 2.
7. Vulnerable contract: It transfers the value of the deposited amount to the attacker's contract.
8. Malicious contract: When a transfer is received, the receive function is called.
9. Malicious contract: The receive function checks if the bank's balance is higher than 1 eth, if yes it calls the withdraw function again on the vulnerable contract.

10. Vulnerable contract: Allows another withdraw because the attacker's balance has not yet been updated.
11. …and so (5-9) on until all 10 eths are pulled out!

Attackers contract:

```
contract LetsRobTheBank {

    VulnerableBank bank;

    constructor (address payable _target) {
        bank = VulnerableBank(_target);
    }

    function attack () public payable {
        bank.deposit{value:1 ether}();
        bank.withdraw();
    }
    function attackerBalance () public view
returns (uint256){
        return address(this).balance;
    }

    receive () external payable {
        if(bank.banksBalance()>1 ether){
            bank.withdraw();
        }
    }
}
```

How to protect yourself against reentrancy attack?

Design functions based on the following principles – Checks Effects Interactions

- First – make all your checks,
- Then – make changes e.g. update balances,
- Finally – call another contract.

CEI pattern will eliminate most of the problems, so try to always build the contract logic based on this scheme. Make all your checks first, then update balances and make changes, and only then call another contract.

Use mutex – add nonReentrant modifier

- Use a ready-made implementation of nonReentrant modifier.
- Add nonReentrant modifier to all external functions.
- Go through all the functions and if you are sure that the vulnerability does not exist in a particular function, remove the modifier.

If for some reason it is not possible / too complicated / expensive to use the CEI pattern. Consider adding a **mutex** to the functions containing external calls. After calling the function, it will be marked as “entered” and lock the state. It will not allow you to call any function that is marked as nonReentrant until the state will be released again.

Remember not to treat **mutexes** as a silver bullet and do not exclude the option for presence of unsafe external calls in the case of integration with other smart contracts.

To protect yourself from design and business logic bugs, the best you can do is constantly raise awareness among your developers and try to prevent vulnerabilities before the code comes out.

A few words at the end

I hope this article has illustrated the basics of the reentrancy attack in a simple way. In the next part, however, we will look in more detail at examples of unsafe external calls that are not so obvious and might have terrible results.

Cross-function and cross-contract unsafe external calls, which are still alive and cause trouble for various projects.

Remember that this is not the end of vulnerabilities in smart contracts. If you have any questions about this article or need support in securing your smart contracts, feel free to use our contact form.

I also encourage you to follow me on twitter, where I regularly publish various tips about the world of smart contracts and more. Stay tuned!