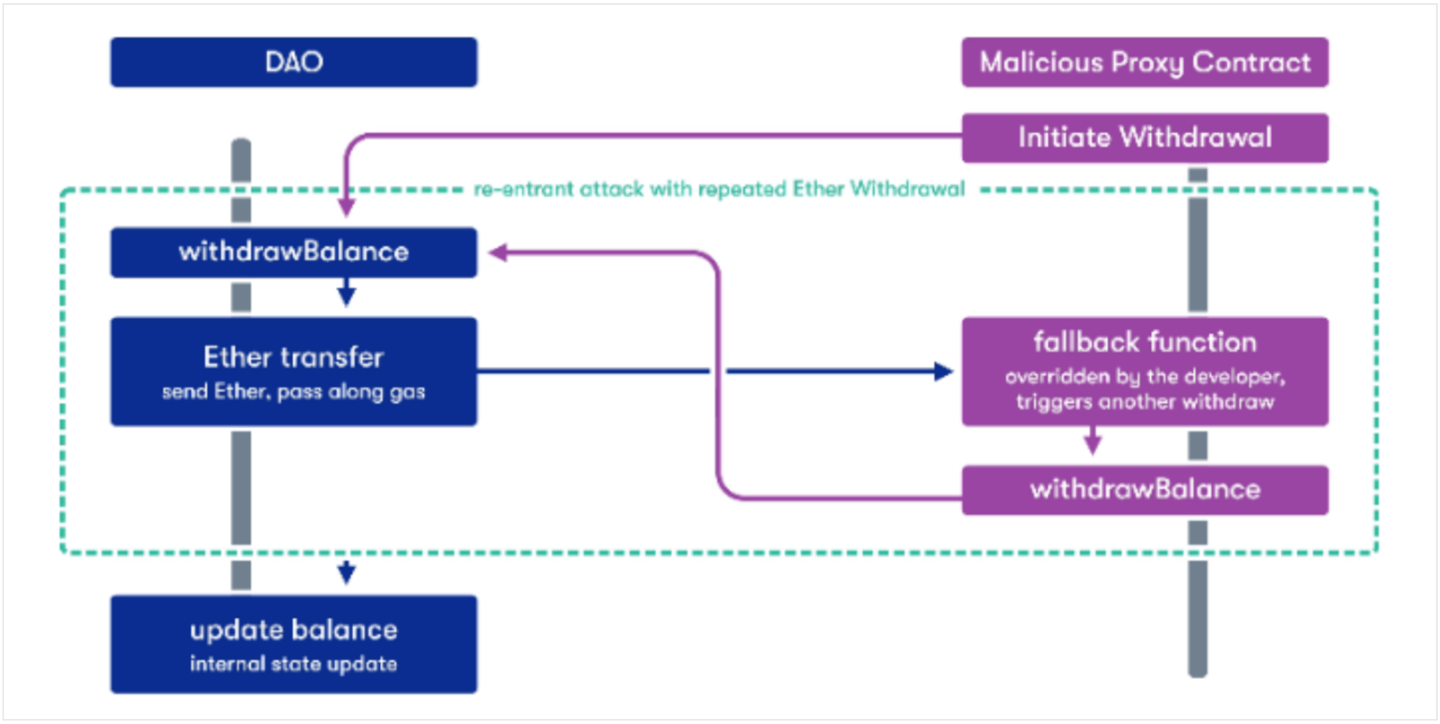


This post is based on an excerpt from Fundamentals of Smart Contract Security. To read more about vulnerabilities and smart contract security issues, buy the book on Amazon or Momentum Press.

Computer scientists say that a procedure is *re-entrant* if its execution can be interrupted in the middle, initiated over (re-entered), and both runs can complete without any errors in execution. In the context of Ethereum smart contracts, re-entrancy can lead to serious vulnerabilities.

The most famous example of this was the DAO Hack, where $70million worth of Ether was siphoned off. More recently, Ethereum's Constaninople hard fork was delayed because a re-entrancy vulnerability was found at the last minute.

So what exactly is a re-entracy vulnerability? How does it work, and how can it be prevented?

Mechanism

An example of a re-entrant process can be sending an e-mail. A user can start typing an e-mail using their favorite client, save a draft, send another e-mail, and finish the message later. This is a harmless example. However, imagine a poorly constructed online banking system for issuing wire transfers where the account balance is checked only at the initialization step. A user could initiate several transfers without actually submitting any of them. The banking system would confirm that the user's account holds a sufficient balance for each individual transfer. If there was no additional check at the time of the actual submission, the user could then submit all transactions and potentially exceed the balance of their account. This is the main mechanism of the re-entrancy exploit which was used in the well-known DAO hack.

Real-world Example - The DAO Hack

The DAO was a popular decentralized investment fund based on smart contracts. In 2016, the DAO smart contract accumulated over $150,000,000 (at the time) of ether. If a project that requested funding received sufficient support from the DAO community, that project's Ethereum address could withdraw ether from DAO. Unfortunately for the DAO, the transfer mechanism would transfer the ether to the external address before updating its internal state and noting that the balance was already transferred. This gave the attackers a recipe for withdrawing more ether than they were eligible for from the contract via re-entrancy.

The DAO hack took advantage of Ethereum's fallback function to perform re-entrancy. Every Ethereum smart contract byte code contains the so-called default fallback function which has the following default implementation shown in Figure 1

This default fallback function can contain arbitrary code if the developer overrides the default implementation. If it is overridden as payable, the smart contract can accept ether. The function is executed whenever ether is transferred to the contract

```
contract EveryContract {

  function () public {
  }
}
```

Fig. 1 The default implementation of the default fallback.

(see the description of methods send(), transfer() and call() below) or whenever a transaction attempts to call a method that the smart contract does not implement.

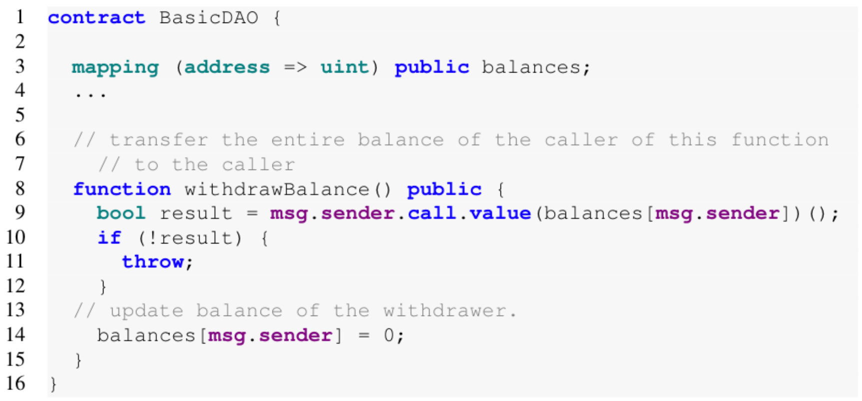
Aside from calling payable methods, Solidity supports three ways of transferring ether between wallets and smart contracts. These supported methods of transferring ether are send(), transfer() and call.value(). The methods differ by how much gas they pass to the transfer for executing other methods (in case the recipient is a smart contract), and by how they handle exceptions. send() and call().value() will merely return false upon failure but transfer() will throw an exception which will also revert state to what it was before the function call. These methods are summarized below

	address.send()	address.transfer()	address.call.value()
Adjustable gas	no	no	yes
Gas limit	2300	2300	all/settable
Behaviour on error	Return false	Throw exception	Return false

Table 1. Methods for transferring ether. By default, all the

remaining gas is available when using call.value(), but the developer may choose to reduce the amount.

In the case of the DAO smart contract (a basic version of which is given in figure 2) the ether was transferred using the call.value() method. That allowed the transfer to use the maximum possible gas limit and also prevented reverting the state upon possible exceptions. Thus, the attackers were able to create a sequence of recursive calls to siphon off funds from the DAO using a smart contract similar to the one presented in Figure 3

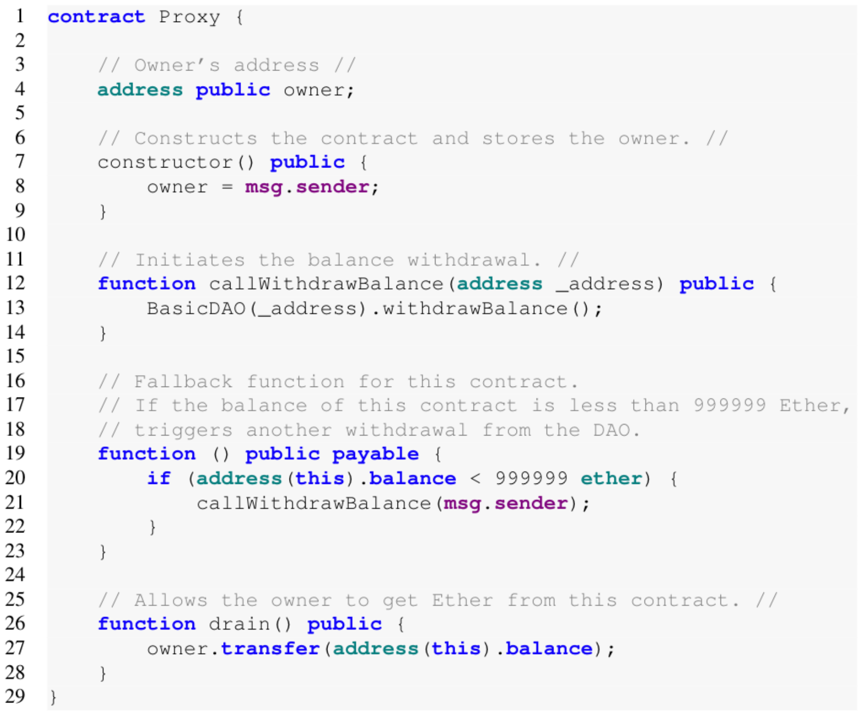


```
contract BasicDAO {

  mapping (address => uint) public balances;
  ...

  // transfer the entire balance of the caller of this function
    // to the caller
  function withdrawBalance() public {
    bool result = msg.sender.call.value(balances[msg.sender])();
    if (!result) {
      throw;
    }
  // update balance of the withdrawer.
    balances[msg.sender] = 0;
  }
}
```

Fig.2 A simplified DAO contract with reentrancy vulnerability in its withdrawBalance() function.



```
contract Proxy {

    // Owner's address //
    address public owner;

    // Constructs the contract and stores the owner. //
    constructor() public {
        owner = msg.sender;
    }

    // Initiates the balance withdrawal. //
    function callWithdrawBalance(address _address) public {
        BasicDAO(_address).withdrawBalance();
    }

    // Fallback function for this contract.
    // If the balance of this contract is less than 999999 Ether,
    // triggers another withdrawal from the DAO.
    function () public payable {
        if (address(this).balance < 999999 ether) {
            callWithdrawBalance(msg.sender);
        }
    }

    // Allows the owner to get Ether from this contract. //
    function drain() public {
        owner.transfer(address(this).balance);
    }
}
```

Fig. 3 A smart contract for exploiting re-entrancy in BasicDAO from Figure 2

The result was the following sequence of actions (also depicted below in Figure 4):

1. The proxy smart contract would ask for a legitimate withdrawal.
2. The transfer from BasicDAO to the proxy smart contract triggered a fallback function.

3. The proxy smart contract fallback function would ask BasicDAO for another withdrawal.
4. The transfer from BasicDAO to the proxy smart contract triggered a fallback function.
5. The proxy smart contract fallback function would ask BasicDAO for another withdrawal.
6. . . .

Note that the balance of the proxy smart contract was never updated (this happens after the transfer). Furthermore, notice that unless the transfer to the proxy contract fails, an exception is never thrown and the state never gets reverted.

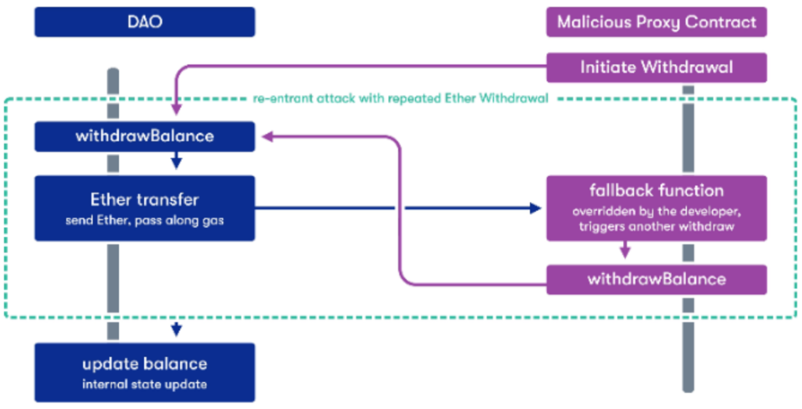


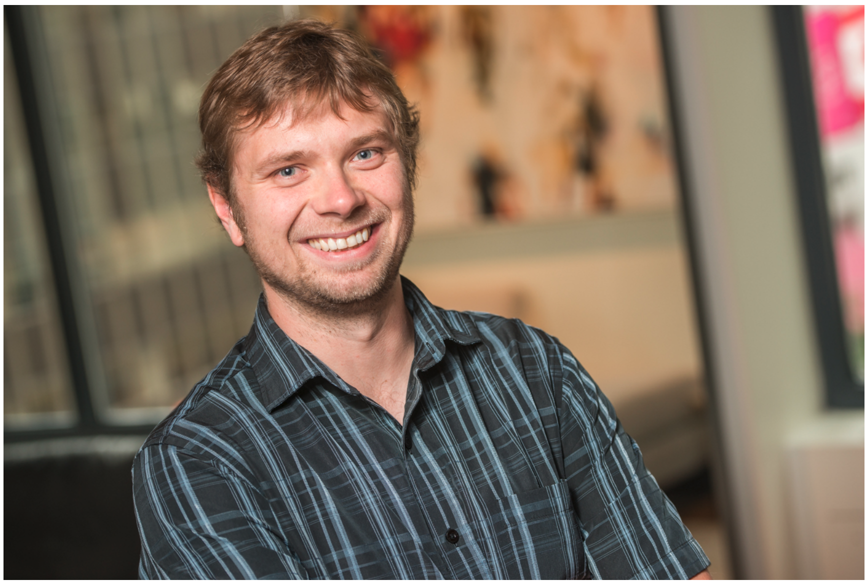
Fig. 4 An illustration of the re-entrancy attack.

Prevention

The re-entrancy attack in the DAO contract could have been avoided in several ways. Using the functions send() or transfer() instead of call.value() would not allow for recursive withdrawal calls due to the low gas stipend. Manually limiting the amount of gas passed to call.value() would achieve the same result.

Yet, there is a much simpler best practice that makes any re-entrancy attack impossible. Note that the DAO contract updates the user balance after the ether transfer. If this was done prior to the transfer, any recursive withdraw call would attempt to transfer a balance of 0 ether. This principle applies generally—if no internal state updates happen after an ether transfer or an external function call inside a method, the method is safe from the re-entrancy vulnerability.

About the Author



This post is written by Quantstamp Senior Research Engineer Martin Derka, PhD, and is based on an excerpt from Fundamentals of Smart Contract Security.

Martin holds a Ph.D in Computer Science from the University of Waterloo. He also studied at Brock University, McMaster University, Masaryk University in the Czech Republic, and has additional degrees from some of these. He is former Vanier Scholar and a brief NSERC post-doctoral fellow at Carleton University.

NOVEMBER 11, 2020

NOVEMBER 5, 2020

OCTOBER 28, 2020

OCTOBER 27, 2020