D1: Re-entrancy

Portland State
Computer Science

Where has this term been used before?

#1: Re-entrancy

- **Race to empty, recursive call vulnerability, call to the unknown**
  - Top vulnerability in DASP
  - Calls to external contracts that result in new calls back into the calling contract (often via low-level `call()` that forwards all gas)
  - For the calling function, this means that the contract state may change in the middle of its execution.
- **Loss**: estimated at 3.6M ETH (~$60M at the time)

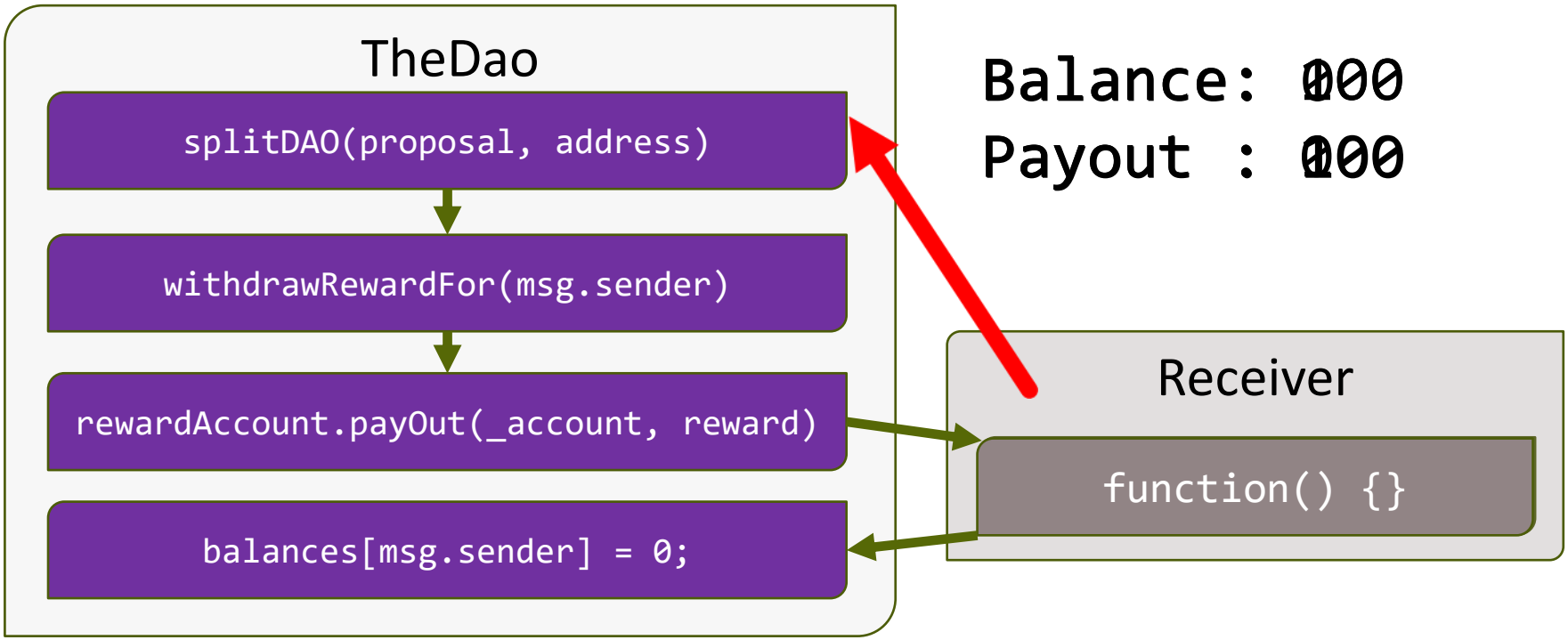
Walkthrough scenario

- A **victim contract** tracks the balance of a number of addresses and allows users to retrieve funds with its public **withdraw()** function.
- A **malicious smart contract** uses the **withdraw()** function to retrieve its entire balance.
- The **victim contract** executes the **call.value(amount)()** **low level function** to send the ether to the **malicious contract** **before updating the balance of the** **malicious contract**.
- The **malicious contract** has a payable **fallback()** function that accepts the funds and then calls back into the **victim contract**'s **withdraw()** function again.
- This second execution triggers a transfer of funds: remember, the balance of the **malicious contract** still hasn't been updated from the first withdrawal.
- The **malicious contract** successfully withdraws its entire balance a second time.

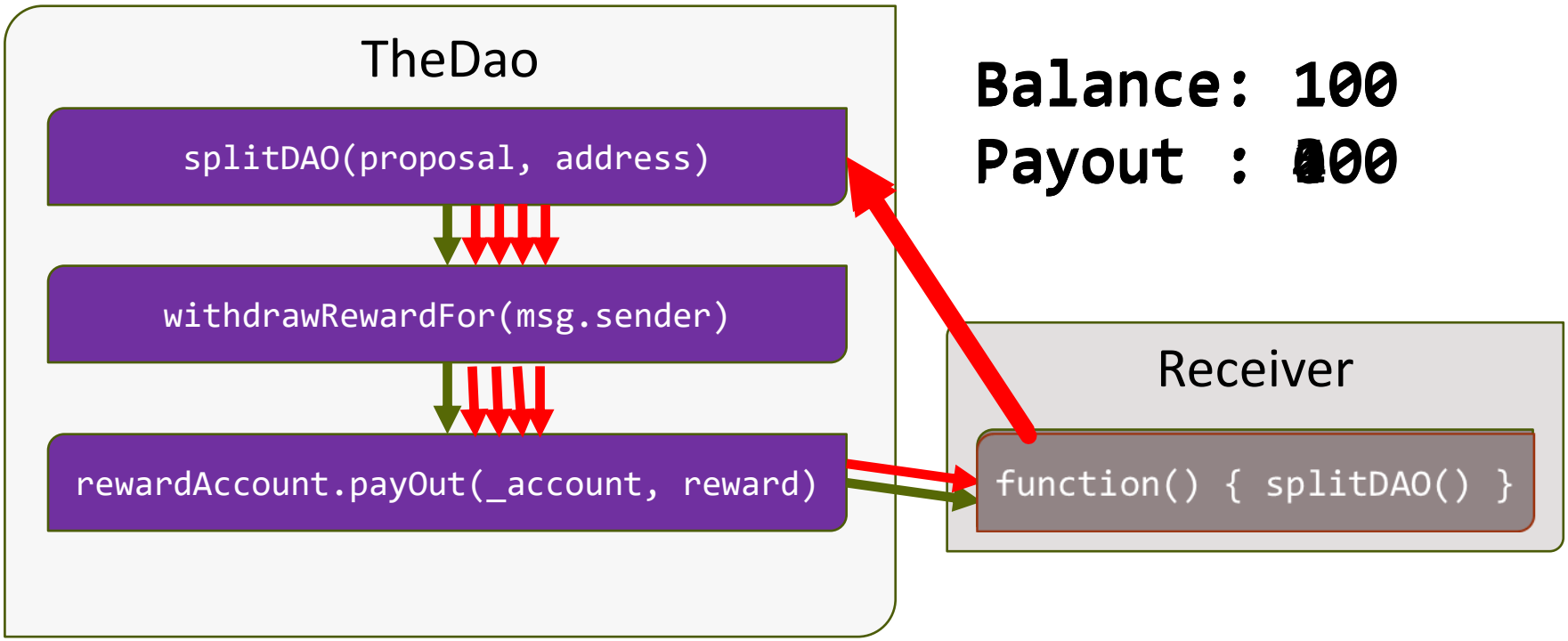
Example #1: DAO

Example #1

- Expected scenario



- Exploitation scenario



- Call before balance update

Example #2: Lendf.me protocol

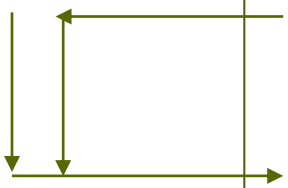
- DeFi (Decentralized Finance) protocol for lending (4/2020)

- Hackers appear to have chained together bugs and legitimate features from different blockchain technologies to orchestrate a sophisticated "reentrancy attack."
- Reentrancy attacks allow hackers to withdraw funds repeatedly, in a loop, before the original transaction is approved or declined.

Code vulnerability example #1

- **withdrawRewardFor()** uses low level **call()** function to send ether to the **msg.sender** address
  - Address is a smart contract and payment will trigger its fallback function with what's left of the transaction gas.
- Fallback function can then call (recurse) back into vulnerable contract to again call **withdrawRewardFor()**
  - Done before balances are updated!

```
// withdrawRewardFor() to get DAO Tokens
if (balances[msg.sender] == 0)
    revert();
withdrawRewardFor(msg.sender);
totalSupply -= balances[msg.sender];
balances[msg.sender] = 0;
paidOut[msg.sender] = 0;
return true;
```



```
function () {
  withdrawRewardFor();
}
```

Remediation #1: Check-effects-interactions

- Vulnerable pattern (check-interactions-effects)

```
function withdraw(uint _amount) {
  require(balances[msg.sender] >= _amount);
  msg.sender.call.value(_amount)();
  balances[msg.sender] -= _amount;
}
```

- Fixed pattern (Checks-effects-interactions)
  - https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html
  - Check all pre-conditions using assert and require
  - Then, make changes to contract state
  - Then, interact with other contracts via external calls

```
function withdraw(uint _amount) {
  require(balances[msg.sender] >= _amount);
  balances[msg.sender] -= _amount;
  msg.sender.call.value(_amount)();
}
```

Check-Effects-Interation

- Counter-intuitive
  - Typical pattern in programming is to apply effects after interactions already have happened
    - Wait for return stating that function execution successful
    - Then change state based on result
  - But, does not need to address multiple encapsulated function invocations (e.g. re-entrancy from within program)
- Must use regardless of trustworthiness of the external call
  - External call my transfer control to a third party that is malicious

```
function getReward(address recipient) public {
    // Check that reward hasn’t already been claimed
    require(!claimedReward[recipient]);

    // Internal work first (claimedReward )
    claimedReward[recipient] = true;

    require(recipient.call.value(rewardValue)());
}
```

```
function buy (uint256 _itemId) payable public {
    require(priceOf(_itemId) > 0);                  // Check
    require(ownerOf(_itemId) != address(0));
    require(msg.value == priceOf(_itemId));
    require(ownerOf(_itemId) != msg.sender);
    require(!isContract(msg.sender));

    address oldOwner = ownerOf(_itemId);
    address newOwner = msg.sender;
    uint256 price = priceOf(_itemId);

    ownerOfItem[_itemId] = newOwner;                // Effects
    priceOfItem[_itemId] = nextPriceOf(_itemId);

    Bought(_itemId, newOwner, price);
    Sold(_itemId, oldOwner, price);

    uint256 cut = 0;
    if (cutDenominator > 0 && cutNumerator > 0) {
        cut = price.mul(cutNumerator).div(cutDenominator);
    }
    oldOwner.transfer(price - cut);                 // Interact
}
```

Remediation #2

- Use a lock/mutex to protect against re-entrancy

```
contract ReentrancyGuard {
    bool private reentrancyLock = false;

    // Prevent contract from calling itself (directly or indirectly).
    modifier nonReentrant() {
        require(!reentrancyLock);
        reentrancyLock = true;
        _;
        reentrancyLock = false;
    }
}
```

  - Modifier then used to protect…

- Malicious contract can not recursively call `claimDay` on transfer

```
function claimDay(uint256 _dayIndex) public nonReentrant payable
{
    ...
    require(msg.sender != seller);
    require(amountPaid >= purchasePrice);
    ...
    // Fire Claim Events
    Bought(_dayIndex, buyer, purchasePrice);
    Sold(_dayIndex, seller, purchasePrice);
    ...
    // Transfer Funds
    if (seller != address(0)) {
        seller.transfer(salePrice);
    }
    if (changeToReturn > 0) {
        buyer.transfer(changeToReturn);
    }
}
```