Blockchain and Smart Contracts
From Theory to Practice

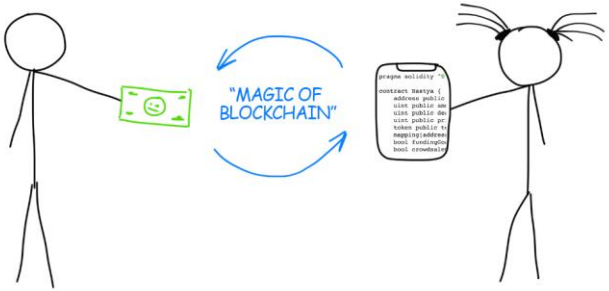
Bruno Rodrigues, Roman Blum, Thomas Bocek and Burkhard Stiller
CSG@IfI, University of Zürich UZH
Hochschule für Technik Rapperswil HSR, Switzerland
[rodrigues¦stiller]@ifi.uzh.ch
[roman.blum¦thomas.bocek]@hsr.ch

University of Zurich[UZH]

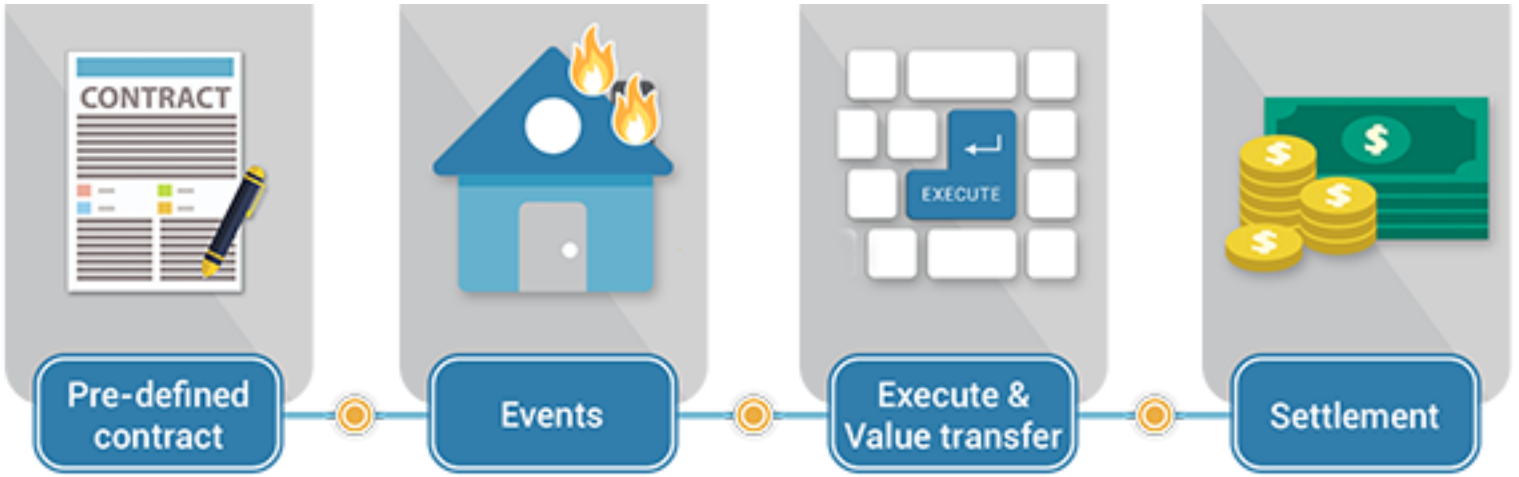
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

What are Smart Contracts?

- First proposed by Nick Szabo in 1994, inventor of “Bit Gold” in 1998
- A computer program that acts as an agreement
- Goal: a superior system for contractual agreements solely based on computer code
- Four basic contract objectives: observability, verifiability, privity and enforceability



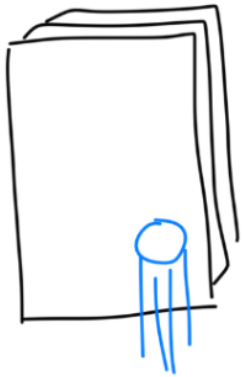
How Do Smart Contracts Work?



- Terms of the policy are agreed by all counterparties
- These are hard coded into the smart contract and cannot be chan-ged without all parties knowing
- Event triggers insurance policy execution
- The smart contract policy is automatically executed based on the pre-agreed terms
- Payout / other settlement completed instantly and efficiently

Traditional vs. Smart Contracts

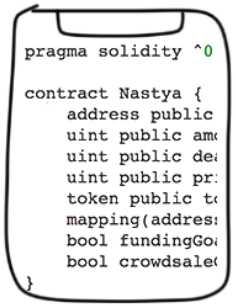
SILLY CONTRACT



- BORING OFFICIAL PAPER
- NO GUARANTEE
- KILL THE TREES
- NEEDS 50 LAWYERS

NOT YOUR BUDDY

SMART CONTRACT



- PERFECT CODE
- VERIFIED BY MATHS
- I'M A PROGRAMMER, YOU CAN'T FOOL ME
- ANYONE CAN WRITE HIS OWN

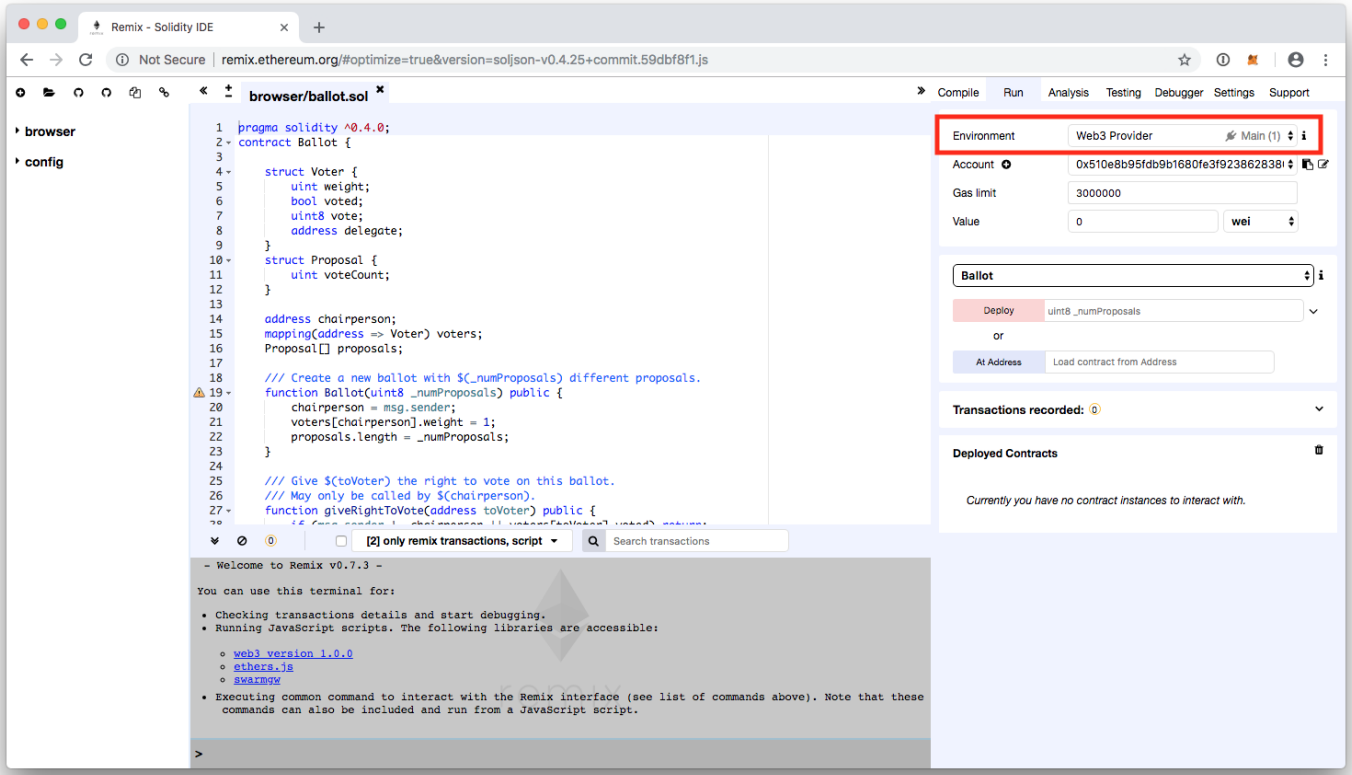
YOUR BUDDY

Let’s Code!

Creating Smart Contracts with Solidity.

Solidity IDE

- http://remix.ethereum.org (Mixed Content: not https)
- Select Web3 Provider (geth is your Web3 Provider)



Solidity - http://solidity.readthedocs.io

- Version Pragma

```
pragma solidity ^0.4.24; // not before 0.4.24, not after 0.5.0
```

- Comments

```
// This is a single-line comment.
/*
This is a
multi-line comment.
*/
```

- Data Types

byte, bytes2 to bytes32, bytes (same as byte[], but more expensive), string, int8 to uint256, address, enum, bool

- Contract

```
contract SimpleStorage {
    uint256 storedData; // State variable
}
```

Solidity IDE

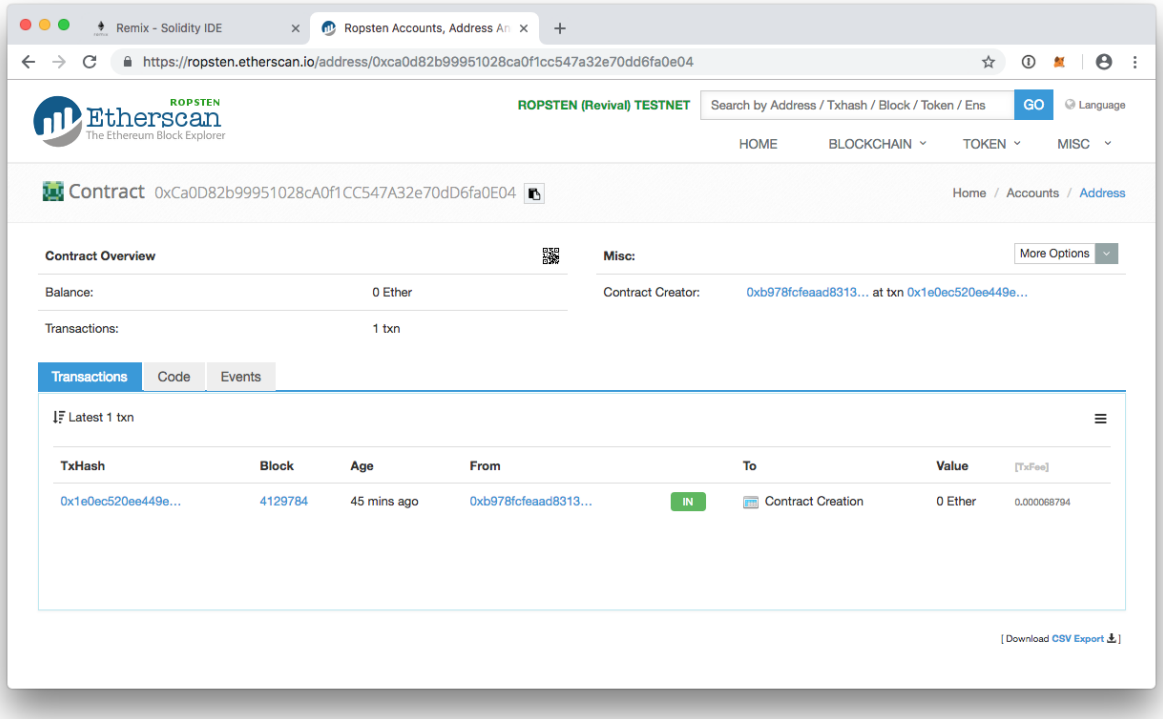
- **Your turn:** Create your first contract

```
pragma solidity ^0.4.24;
// Minimal contract example
contract SimpleStorage {
    uint256 storedData; // State variable
}
```

- Remix – Solidity IDE – Environment: Web3 Provider
  - http://localhost:8545
- Compile and push «Deploy» (red button)

Check Deployment

- You have mined you first contract!



- How much did it cost?

Gas: Ethereum's Fuel

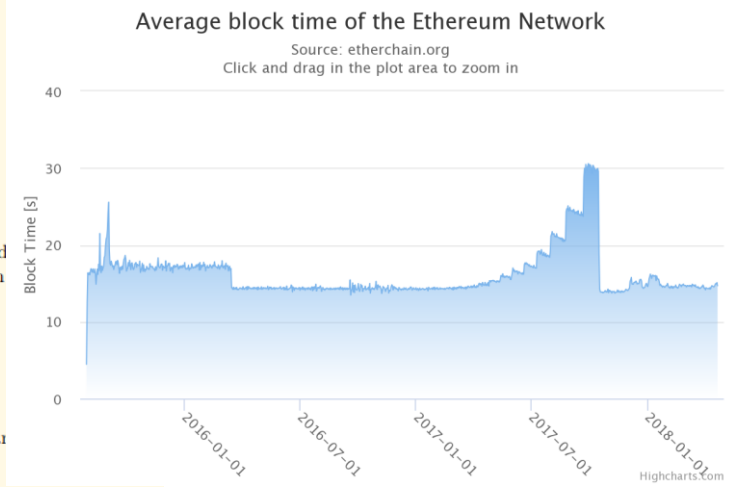
- Price that is paid for running a transaction or a contract
- Unit of measuring computational work
- Similar to the use of Kilowatts (kW) for measuring electricity in your house
- Current Price: https://ethgasstation.info
  Current Gas Price (02.10.18): 8.6 Gwei
- Every instruction needs to be paid for
- If you run out of gas, state is reverted, ETH gone

Gas: Ethereum's Fuel

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value tra
$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA
G_{copy}	3	Partial payment for *COPY operations, multiplied by wo
$G_{blockhash}$	20	Payment for BLOCKHASH operation.



W_{zero} = {STOP, RETURN}
W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, POP, PC, MSIZE, GAS}
$W_{verylow}$ = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}
W_{low} = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}
W_{mid} = {ADDMOD, MULMOD, JUMP}
W_{high} = {JUMPI}
$W_{extcode}$ = {EXTCODESIZE}

Solidity - http://solidity.readthedocs.io

- Structs

```
struct Account {
  string addr;
  uint amount;
}
```

- Mapping (key-value pair, not iterable, but can be implemented), think of `Map<K,V>` in Java or `Dictionary<K,V>` in C#

```
mapping(address => uint256) public balances;
balances[msg.sender] = 100; // Set balance of sender
uint256 balance = balances[msg.sender]; // Get balance of sender
```

Key	Value
0x3f51...	100
0x17aa...	0
0x4eb2...	85
...	...

Solidity – Functions

- «Standard» Functions: Can read and modify the state

```
uint256 counter;
function setCounter(uint256 _newValue) public {
        counter = _newValue;
}
```

- View Functions: Do not modify the state (it's free!)

```
uint256 counter;
function getCounter() view returns (uint256) {
        return counter;
}
```

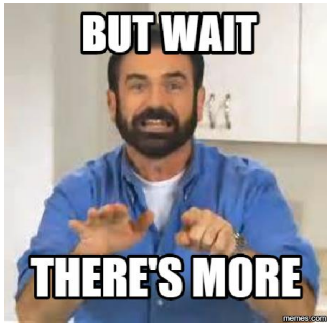
- Pure Functions: Do not read from or modify the state.

```
function multiply(uint256 a, uint256 b) pure returns (uint256) {
        return a * b;
}
```

Solidity – Functions

- Internal Functions: Only callable internally

```
uint256 counter;
function setCounter(uint256 _newValue) internal {
        counter = _newValue;
}
```



- External Functions: Only callable externally

  Note: public and external differs in terms of gas usage

```
function setValues(uint256[20] _values) external {
        // do something
}
```

- Payable Functions: Receives plain Ether

```
function deposit() payable {
        // Access msg.value to get amount of Ether
}
```

Solidity – Basics

Creating a smart contract that can get or set the balance of an address the *wrong* way.

```
contract BankExample {
  mapping(address => uint256) private balances;

  function getBalance(address _address) view public returns (uint256) {
    return balances[_address];
  }

  function setBalance(uint256 _newBalance, address _address) public {
    balances[_address] = _newBalance;
  }
}
```

Solidity – Basics

- Special Variables and Global Functions
  - `block.number` (type of uint256): current block number
  - `block.difficulty` (type of `uint256`): current block difficulty
  - `block.gaslimit` (type of `uint256`): current block gaslimit
  - `gaslet() returns (uint 256)`: remaining gas
  - `msg.sender` (type of `address`): sender of the message
  - `msg.value` (type of `uint256`): number of wei sent with the message
  - `now` (type of `uint256`): current block timestamp

Your turn: Update the previous smart contract such that only the sender of a transaction can get and set his own balance.

Solidity – Basics

Solution: Use the `msg.sender` global variable to let only the sender of the transaction set his own balance.

```
contract BankExampleV2 {
  mapping(address => uint256) private balances;

  function getBalance() public view returns (uint256) {
    return balances[msg.sender];
  }

  function setBalance(uint256 _newBalance) public {
    balances[msg.sender] = _newBalance;
  }
}
```

Solidity – Ownership

- Who is the owner of a smart contract?
- How to transfer ownership?
- The *wrong* way:

```
contract OwnedContract {
   address owner;
   constructor() public {
     owner = msg.sender;
   }

   function transfer(address _newOwner) public {
     owner = _newOwner;
   }
 }
```

Solidity – Basics

- Everybody can transfer the ownership
- Use `require` to test user inputs

```
if (msg.sender != owner) { throw; }
// Do something only the owner is allowed to
```

behaves exactly the same as

```
require(msg.sender == owner, "Unauthorized");
// Do something only the owner is allowed to
```

- The use of revert(), assert(), and require() in Solidity

Solidity – Ownership

```
contract OwnedContract {
   address owner;
   constructor() public {
     owner = msg.sender;
   }

   function transfer(address _newOwner) public {
     require(owner == msg.sender, "Sender not authorized");
     owner = _newOwner;
   }
}
```

Your turn: Update the previous BankExample contract to let the owner set balances arbitrarily. Hint: Create a second function to set the balance, e.g.

```
setBalance(uint256 _balance, address _address
```

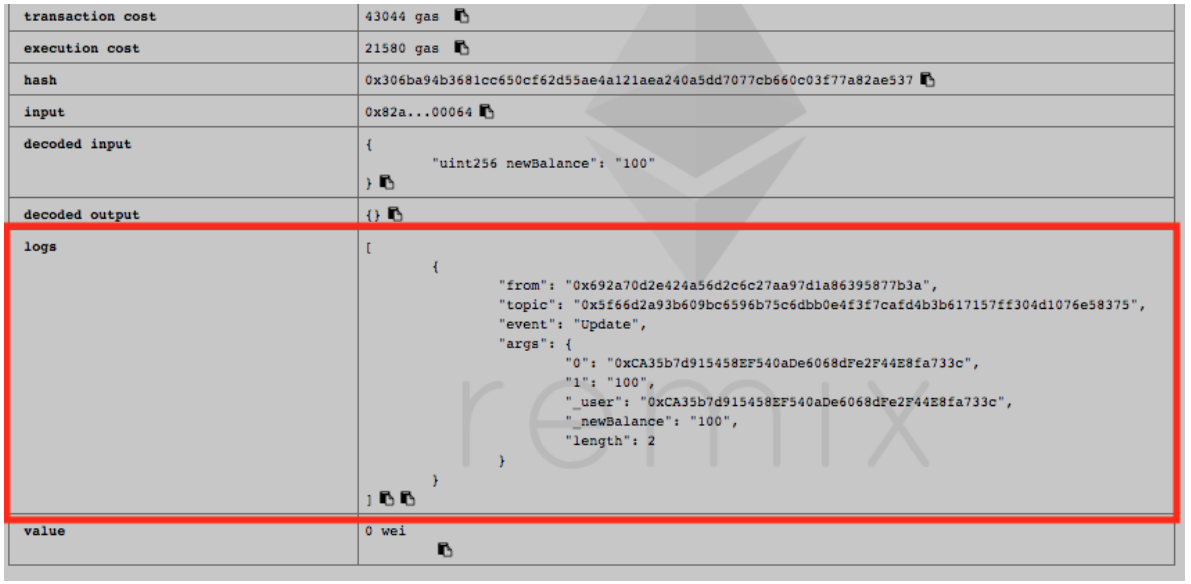
Solidity – Ownership

Solution: Create a second `setBalance` method which uses `require` to verify if the sender of the transaction is the owner.

```
contract OwnedBankExample {
  ...

  function setBalance(uint256 _newBalance) public {
    balances[msg.sender] = _newBalance;
  }

  function setBalance(uint256 _newBalance, address _address) public {
    require(owner == msg.sender, "Sender not authorized");
    balances[_address] = _newBalance;
  }
}
```

Solidity – Events/Notifications

- Events are a way for smart contracts written in Solidity to log that something has occurred
- Interested observers, notably JavaScript front ends for decentralized apps, can watch for events and react accordingly.



Solidity – Events/Notifications

```
contract EventsExample {
   event OwnerChanged(
     address _oldOwner,
     address _newOwner
   );

   function transfer(address _newOwner) public {
     require(owner == msg.sender, "Sender not authorized");
     emit OwnerChanged(owner, _newOwner);
     owner = _newOwner;
   }
}
```

Your turn: Update the previous BankExample. Emit an event when the balance of an address changes. The event should include the address, the old balance and the new balance.

Solidity – Events/Notifications

Solution:

```
event BalanceChanged(address _address,
                     uint256 _oldBalance, uint256 _newBalance);

function setBalance(uint256 _newBalance) public {
   uint256 _oldBalance = balances[msg.sender];
   balances[msg.sender] = _newBalance;
   emit BalanceChanged(_address, _oldBalance, _newBalance);
}

function setBalance(uint256 _newBalance, address _address) public {
  require(owner == msg.sender, "Sender not authorized");
  uint256 _oldBalance = balances[_address];
  balances[_address] = _newBalance;
  emit BalanceChanged(_address, _oldBalance, _newBalance);
}
```

Solidity – Events/Notifications

Better Solution:

```
event BalanceChanged(address indexed _address,
                     uint256 _oldBalance, uint256 _newBalance);

function setBalance(uint256 _newBalance) public {
  _setBalance(_newBalance, msg.sender);
}
function setBalance(uint256 _newBalance, address _address) public {
  require(owner == msg.sender, "Sender not authorized");
  _setBalance(_newBalance, _address);
}

function _setBalance(uint256 _newBalance, address _address) internal {
  uint256 _oldBalance = balances[_address];
  balances[_address] = _newBalance;
  emit BalanceChanged(_address, _oldBalance, _newBalance);
}
```

ERC20 Token Standard

- Describes the functions and events that an Ethereum token contract has to implement

```
contract ERC20 {
    function totalSupply() public view returns (uint256 totalSupply);
    function balanceOf(address _owner) public view returns (uint256 balance);
    function transfer(address _to, uint _value) public returns (bool success);
    function transferFrom(address _from, address _to, uint _value)
        public returns (bool success);
    function approve(address _spender, uint _value)
        public returns (bool success);
    function allowance(address _owner, address _spender) public view
        returns (uint remaining);
    event Approval(address indexed _owner, address indexed _spender, uint _value);
    event Transfer(address indexed _from, address indexed _to, uint _value);
}
```

- Understanding ERC20 Token Contracts

ERC20 Token Standard

- Our token: only partial ERC20 support

```
contract ERC20 {
    function totalSupply() public view returns (uint256 totalSupply);
    function balanceOf(address _owner) public view returns (uint256 balance);
    function transfer(address _to, uint _value) public returns (bool success);
    function transferFrom(address _from, address _to, uint _value)
        public returns (bool success);
    function approve(address _spender, uint _value)
        public returns (bool success);
    function allowance(address _owner, address _spender) public view
        returns (uint remaining);
    event Approval(address indexed _owner, address indexed _spender, uint _value);
    event Transfer(address indexed _from, address indexed _to, uint _value);
}
```

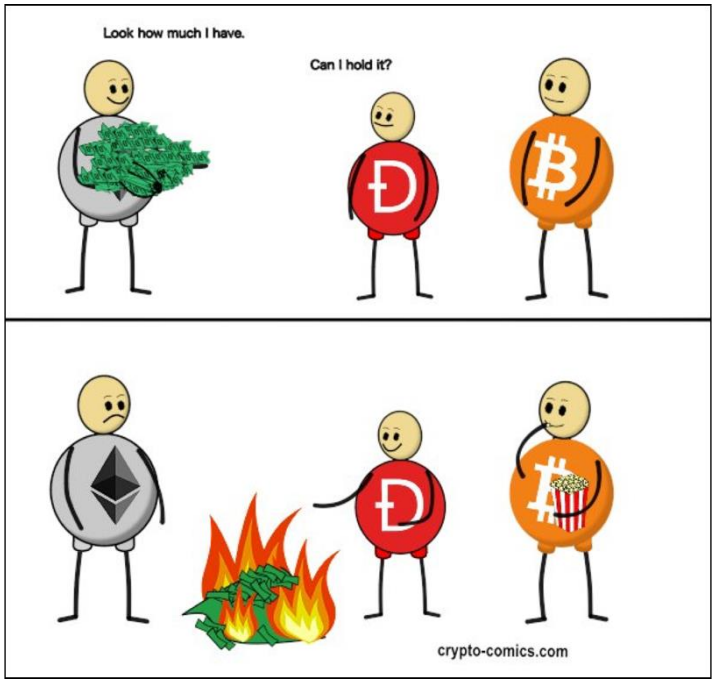
Your turn!

```
pragma solidity ^0.4.24;
contract YourTestToken {
    string public constant name = "Your Token Name";
    string public constant symbol = "YTN";
    function totalSupply() constant returns (uint256 totalSupply) {
        // Your code
    }
    function balanceOf(address _owner) constant returns (uint256 balance) {
        // Your code
    }
    function transfer(address _to, uint256 _value) returns (bool success) {
        // Your code
    }
}
```

- Hint: you only need three state variables (not public):
  - `address owner;`
  - `mapping (address => uint) accounts;`
  - `uint256 totalSupply;`

Smart Contract Security

- Smart contract programming requires a different engineering mindset
- Cost of failure can be high, and change can be difficult
- Notable Examples: Parity Bug, DAO Hack
- DASP: Collaborative project to join efforts in discovering smart contract vulnerabilities, see https://dasp.co/index.html



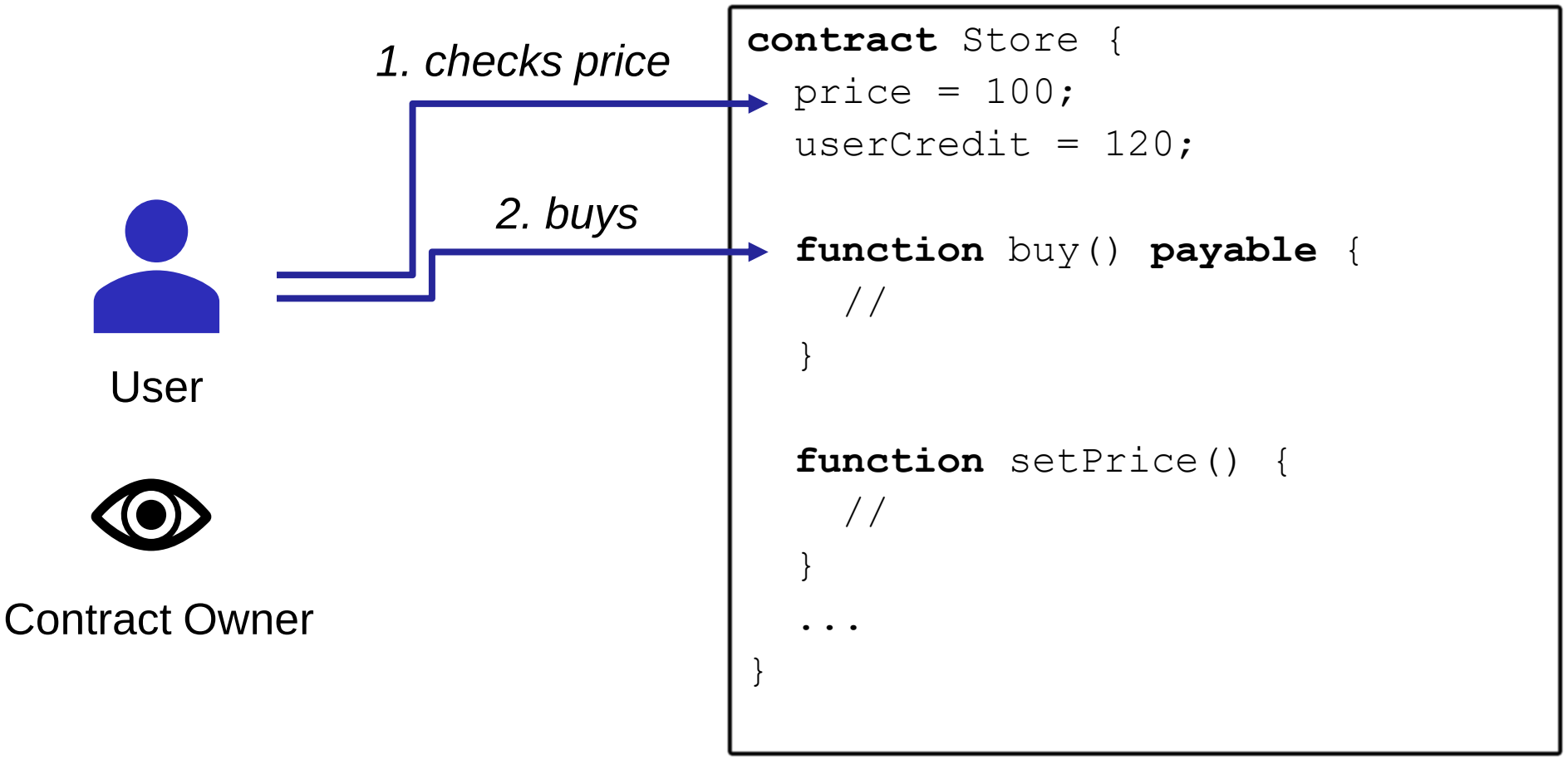
Smart Contract Security

- **Transaction Ordering**
  > Blockchain Shop

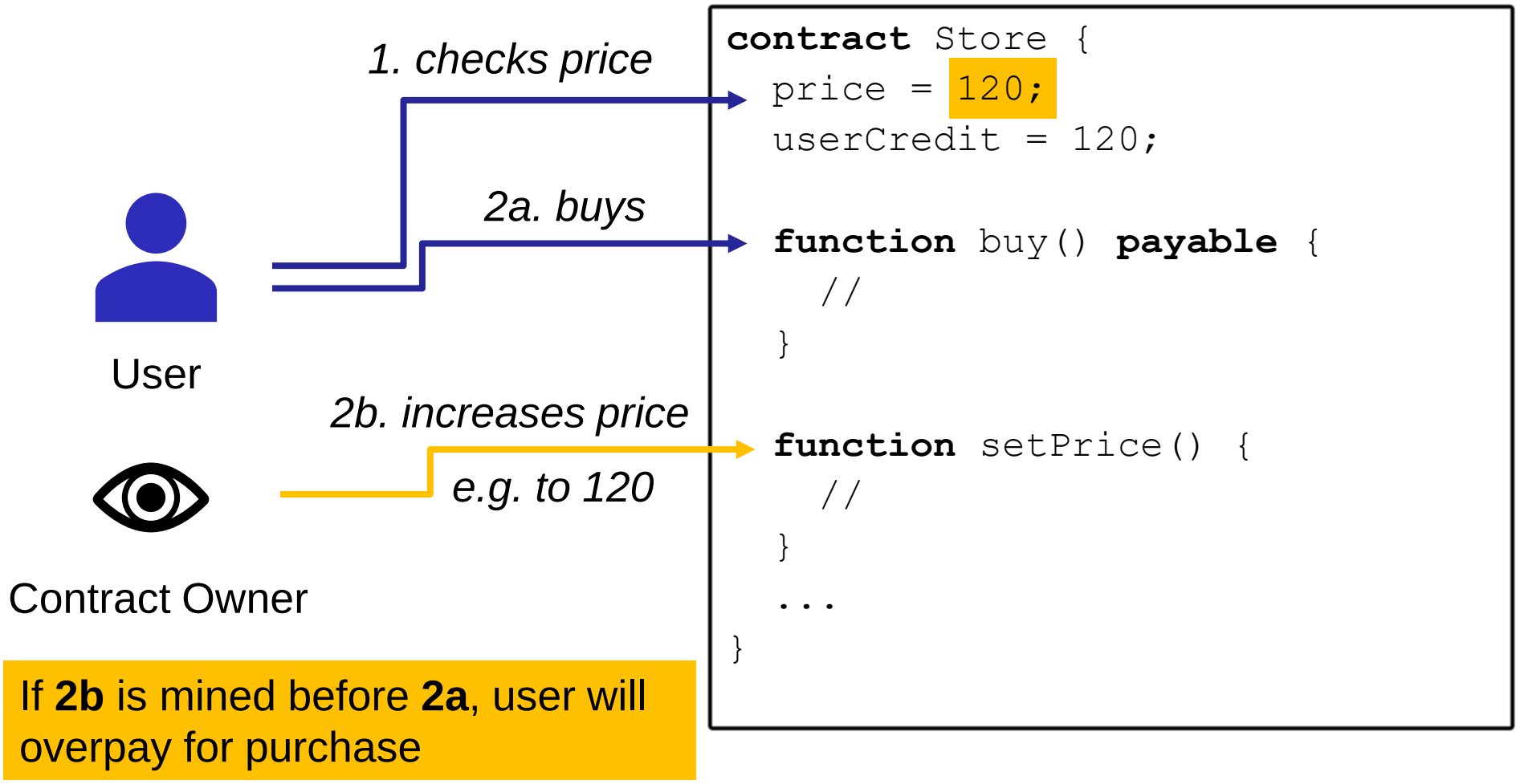
- **Reentrancy Attacks**
  > Good ATM | Bad ATM

Source: James Chiang

Transaction Ordering



Transaction Ordering



If **2b** is mined before **2a**, user will overpay for purchase

Transaction Order Guard

User

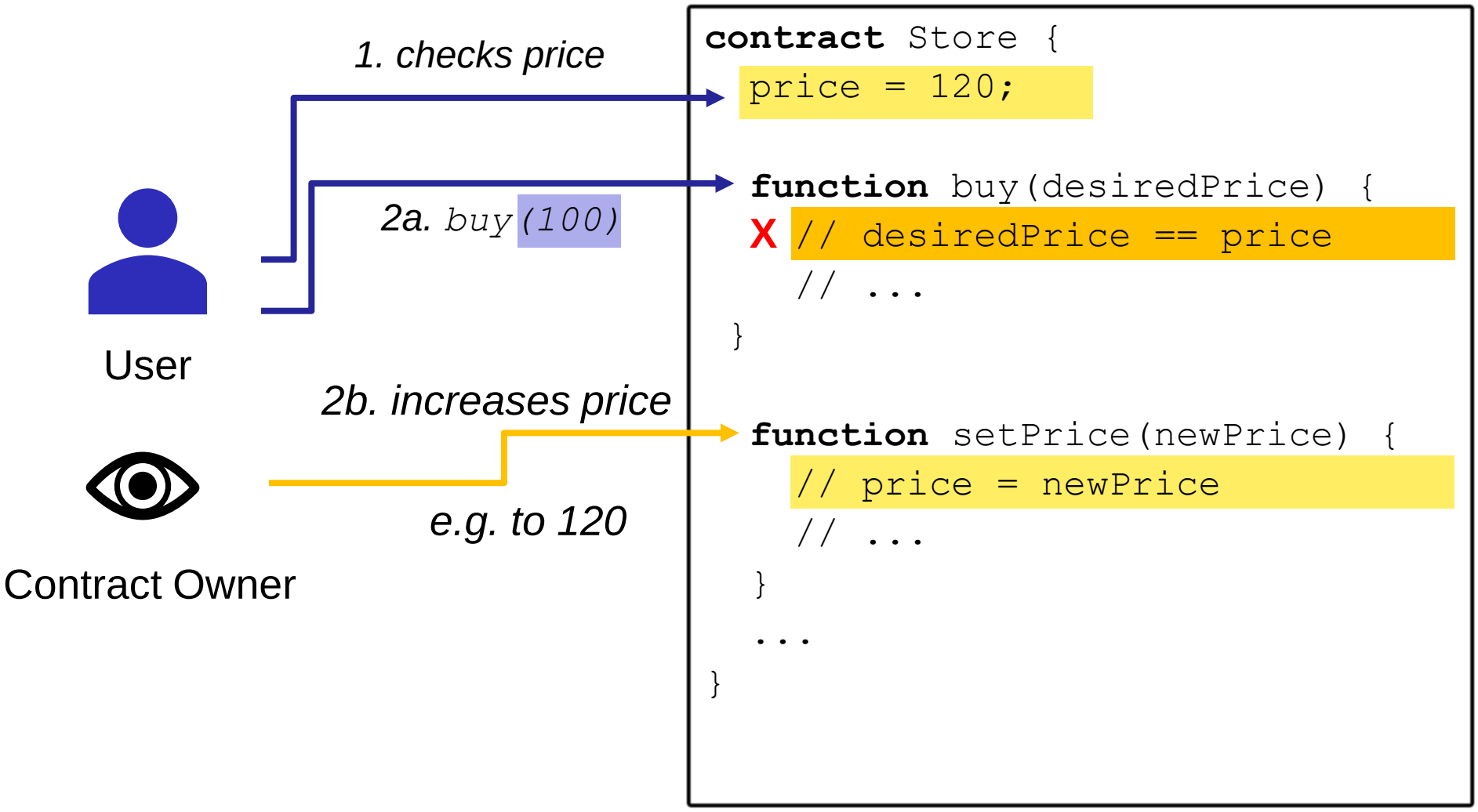
Contract Owner

1. checks price

2. buy(100)

```
contract Store {
  price = 100;

  function buy(desiredPrice) {
    // desiredPrice == price
    // ...
  }

  function setPrice(newPrice) {
    // price = newPrice
  }
  ...
}
```

Transaction Order Guard

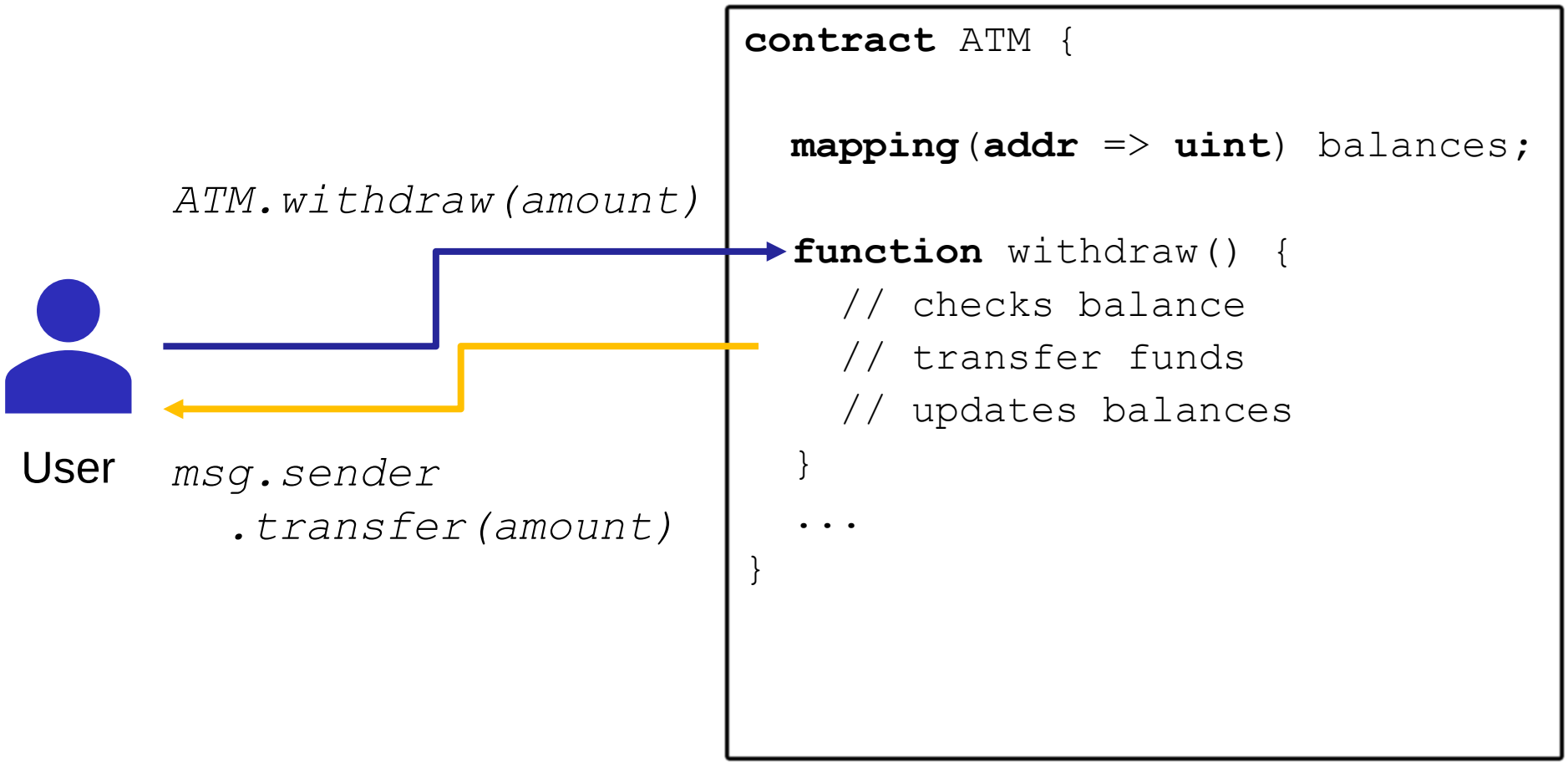


Smart Contract Security

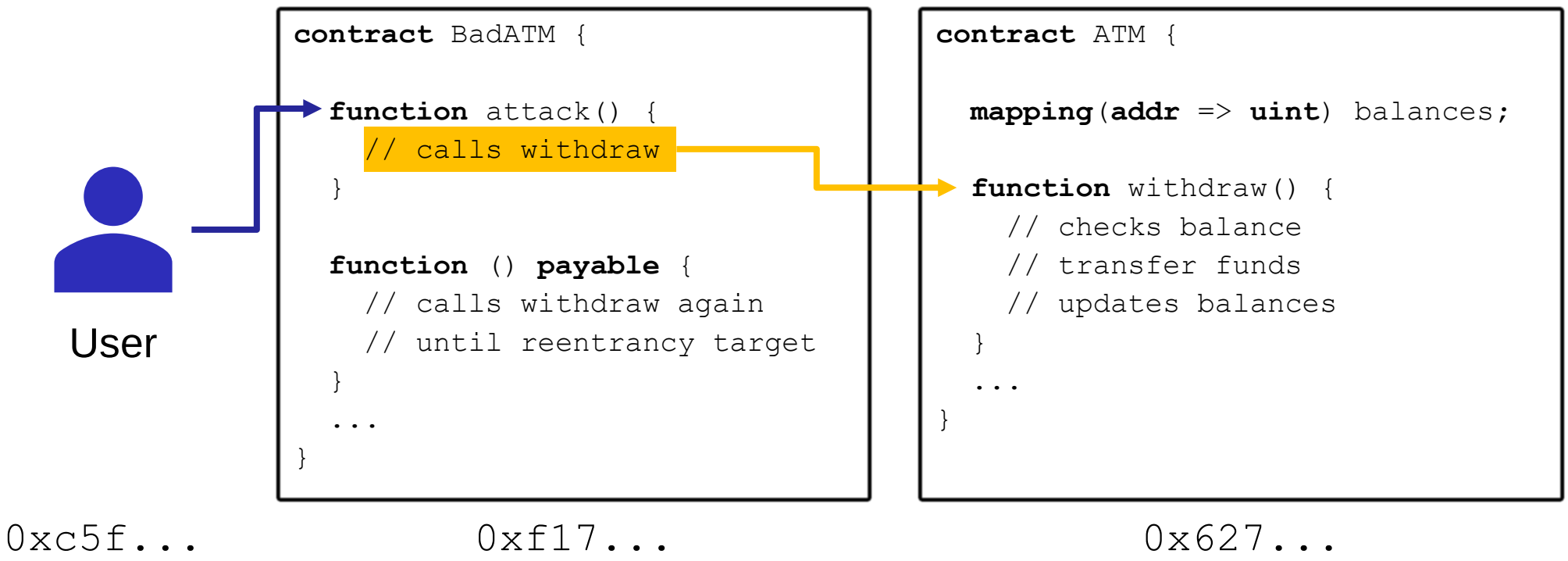
- **Transaction Ordering**
  > Blockchain Shop

- **Reentrancy Attacks**
  > Good ATM | Bad ATM

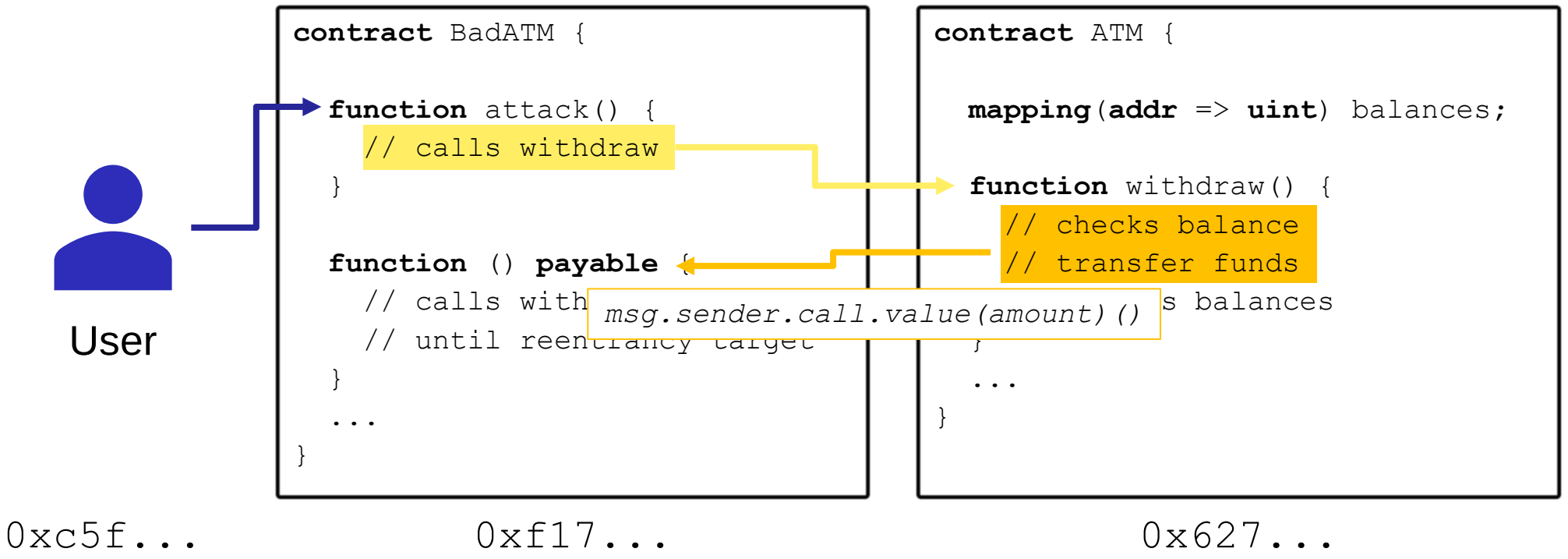
ATM Contract



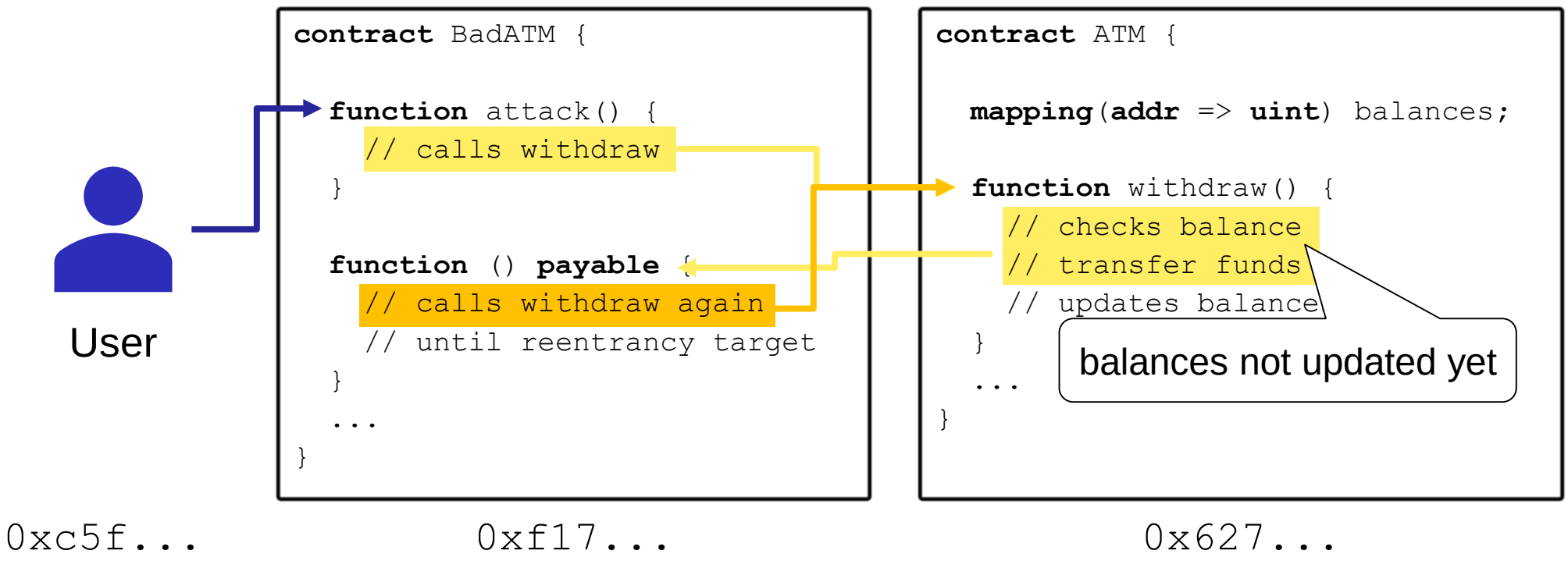
Reentrancy Attack



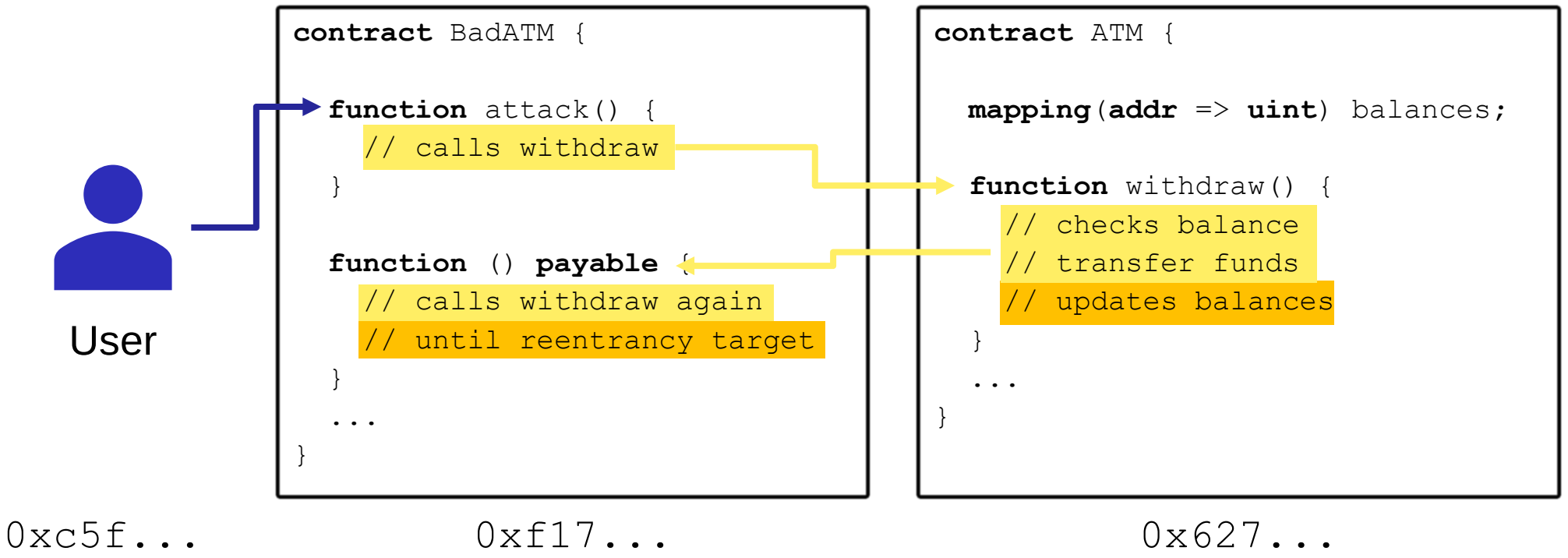
Reentrancy Attack



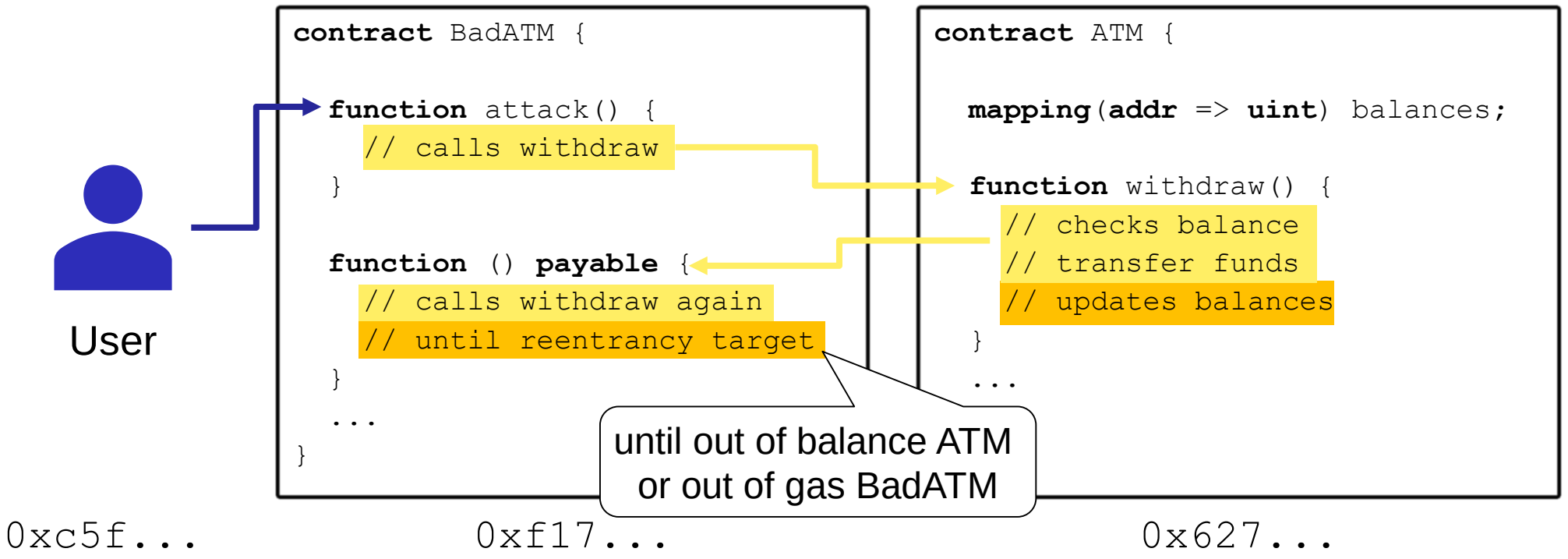
Reentrancy Attack



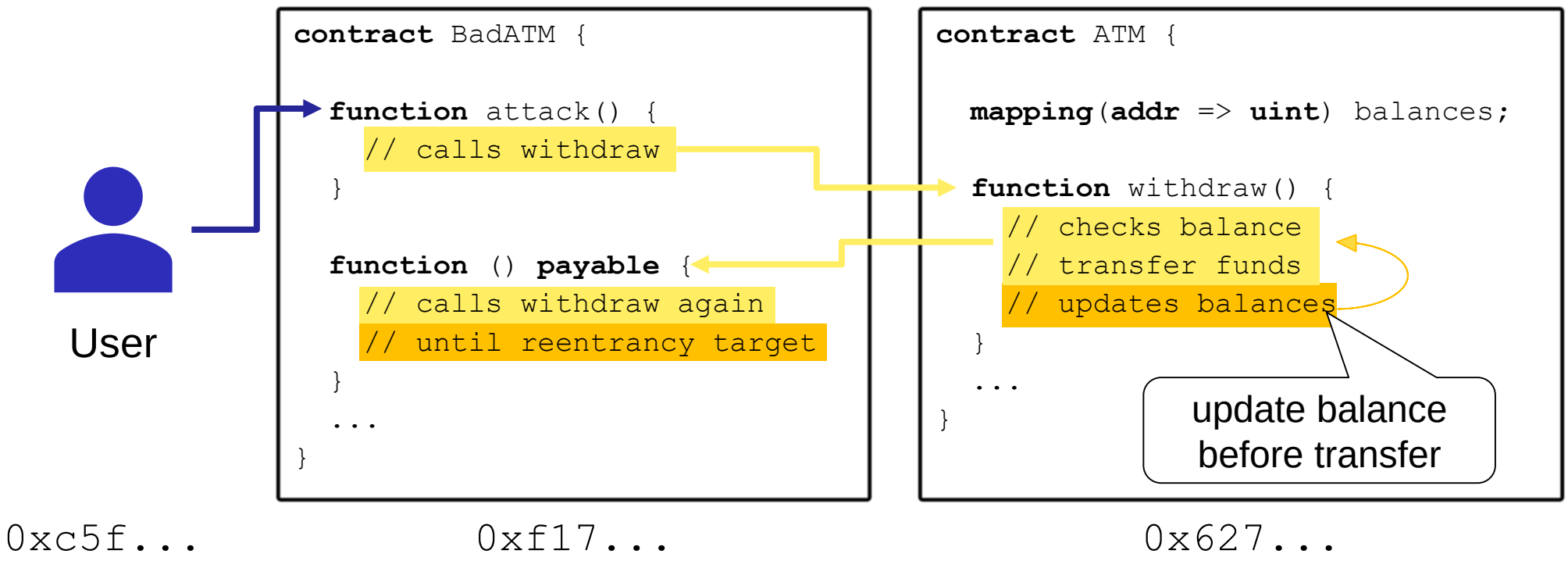
Reentrancy Attack



Reentrancy Attack



Reentrancy Attack



Contract Locks

User

0xc5f...

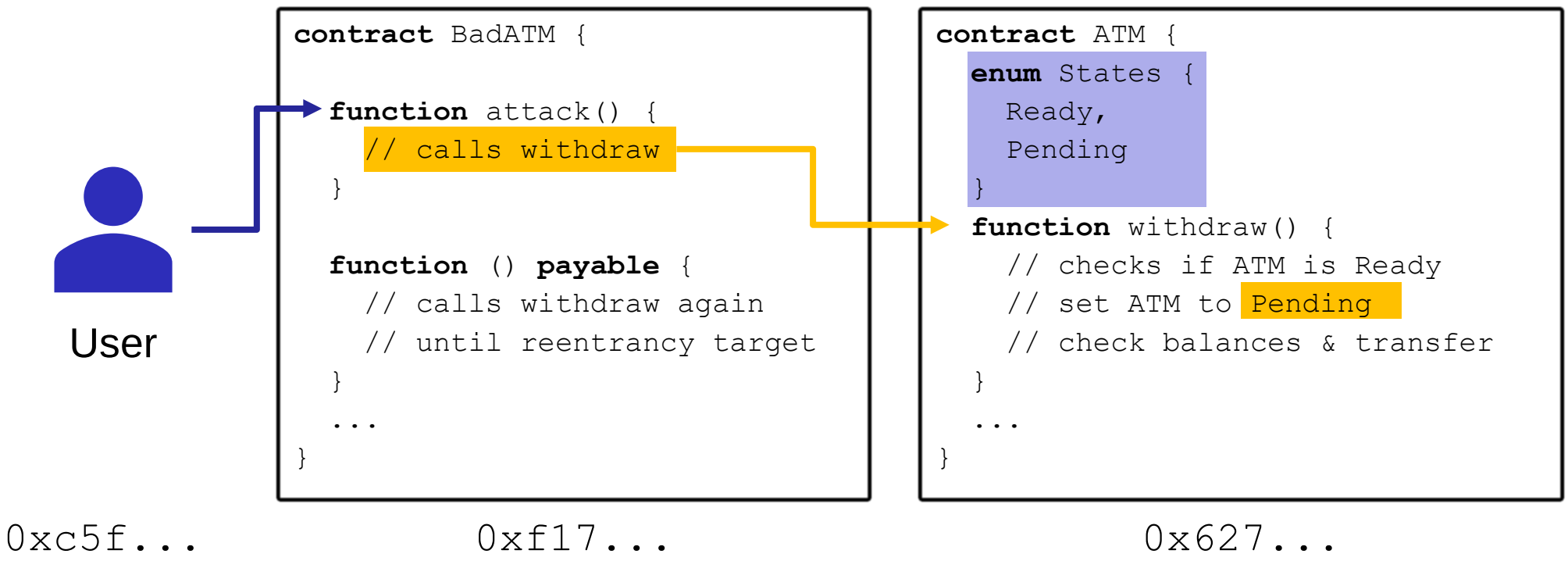
```
contract BadATM {

  function attack() {
    // calls withdraw
  }

  function () payable {
    // calls withdraw again
    // until reentrancy target
  }
  ...
}
```

0xf17...

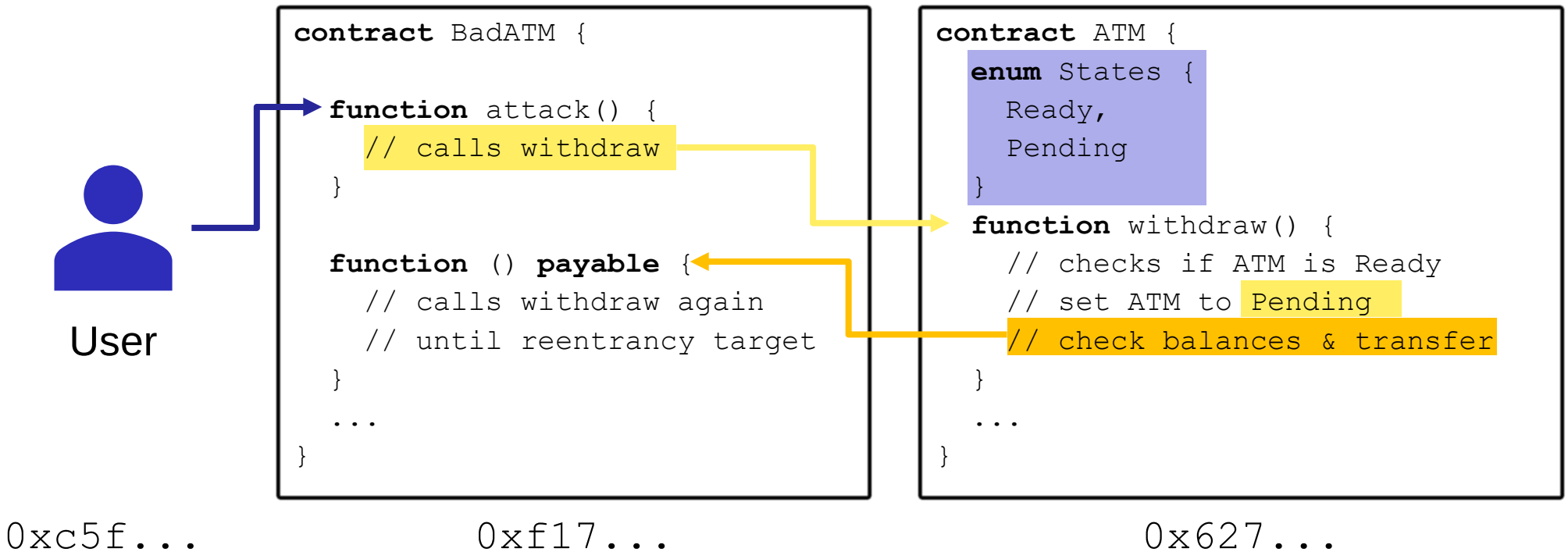
```
contract ATM {
  enum States {
    Ready,
    Pending
  }
  function withdraw() {
    // checks if ATM is Ready
    // set ATM to Pending
    // check balances & transfer
  }
  ...
}
```

0x627...

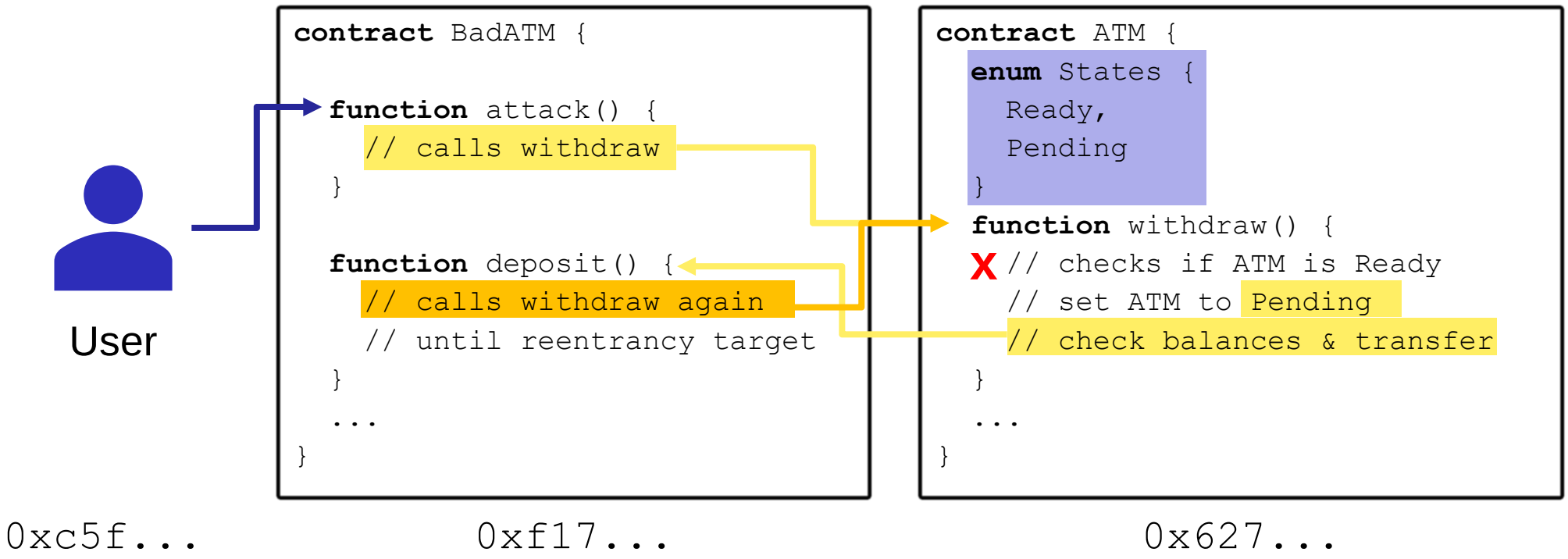
Contract Locks



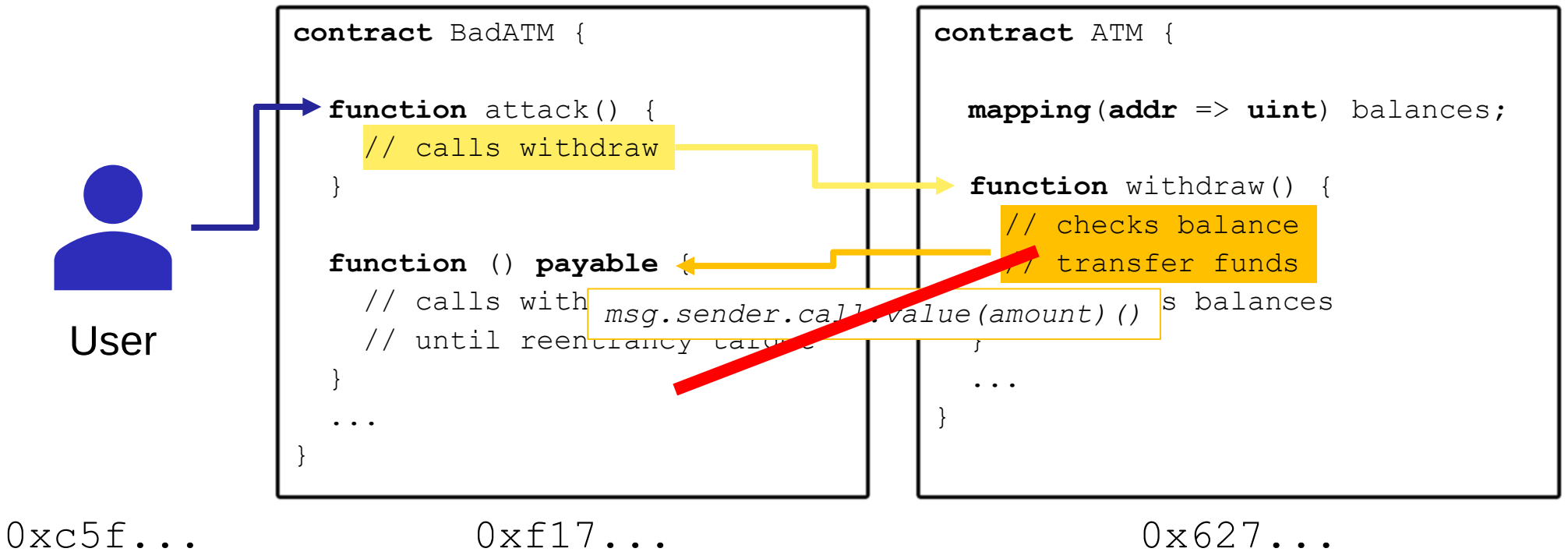
Contract Locks



Contract Locks

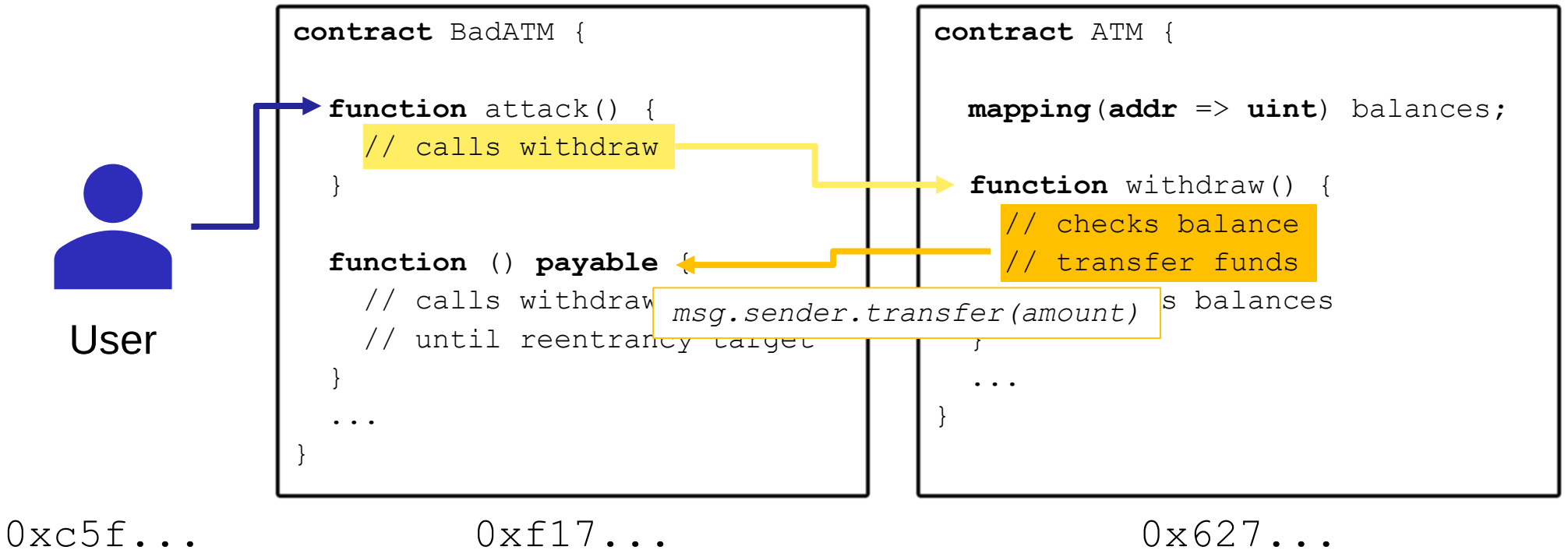


Reentrancy Attack



The transfer-Function

- Only a small amount of gas is sent along (21,000 gas).
- The receiver can only emit one single event at max, safe by "accident".



In Conclusion - Best Practices

- Prepare for failure
- Rollout carefully
- Keep contracts simple
- Stay up to date
- Be aware of blockchain properties

Others

- Safe Math (Overflow)
- Error Handling (Revert / Require / Throw)
- Best Practices e.g. Recommendation for Smart Contract Security in Solidity

Thank You for Your Attention!

Questions?