BLOCKCHAIN: SMART CONTRACTS LECTURE 8 – **SECURITY IN ETHEREUM**

FLORIN CRACIUN

IMPORTANT

Some of the following slides are the property of

Dr. Emanuel Onica & Dr. Andrei Arusoaie

Faculty of Computer Science,

Alexandru Ioan Cuza University of Iași

and are used with their consent.

CONTENTS

THE VERY BASICS – ACCOUNT SECURITY

- **Each EOA is associated with a public/private key pair**
- **At EOA creation:**
  - **JSON keyfile holding the random generated key pair**
  - **Private key encrypted with a password provided by user**
  - **Transactions require private key => require password**

THE VERY BASICS – ACCOUNT SECURITY

- **Key files typically stored in .keystore directory:**


```
{"address":"14fd75ee3ba1acdeb7d361615b277eb11f3d376e",
 "crypto":
        {"cipher":"aes-128-ctr",
        "ciphertext":"c4d7c0b397094cc5c31856b3118f97f45f86fdec73faf1471fdd44108298f08e",
        "cipherparams":
                {"iv":"4ea6e15e7567bbef4a7fddd86f304def"},
        "kdf":"scrypt",
        "kdfparams":{
                "dklen":32,
                "n":262144,
                "p":1,
                "r":8,
                "salt":"e3b1d24871afd890538b90457a25c99758ac610dd426d39e84ae0e4cfd8d0709"},
        "mac":"929dd7c56801596b9d508ba438928786cc7e07654d816e1c38201b820687e4ab"},
 "id":"3fc3b651-dc5a-43cd-a57d-f95936d0ebb5",
 "version":3}
```

THE VERY BASICS – ACCOUNT SECURITY

Steps of encrypting the private key:

- A KDF derivation function generates a derived key using the *kdfparams*:

$$derived\_key = kdf(password, kdfparams)$$

- The encryption key is extracted from the derived_key, i.e., for AES-128, the first 16 bytes:

$$enc\_key = derived\_key[:16]$$

- Encryption is performed using the specified cipher and parameters:

$$ciphertext = aes\text{-}128\text{-}ctr(private\_key, enc\_key, cipherparams)$$

THE VERY BASICS – ACCOUNT SECURITY

- A *mac* is computed for integrity check purposes using the second part of the derived key

$$mac = sha3(derived\_key[16:32] + ciphertext)$$

- The password is used to re-generate the derived key to decrypt the private key using reversed steps
- The *mac* is re-computed and compared to the stored one to check the integrity

SOLIDITY – SECURITY PATTERNS

Access Control

- Purpose:
  a) restrict functionality access based on specific conditions
  b) restrict access to contract state data
- Implementation:
  a) modifiers and Guard Check pattern
  b) access modifiers – *private*

But for b) contract data is public by nature as part of the blockchain structure...

Precise purpose: *private* modifiers prevent a direct programatical access <u>from other contracts</u>

SOLIDITY – SECURITY PATTERNS

Checks-Effects-Interaction

- Purpose: avoid *re-entrancy* based attacks
- Re-entrancy based attack:
  - execution control flow is transferred from a victim contract to a malicious contract typically via a *call/delegatecall*
  - the malicious contract attempts re-executing the initial function in the victim contract with malicious purpose
- Implementation:
  1) do *checks* first to enforce conditions on avoiding re-entry
  2) perform *effects* second: a stale state can lead to re-entry
  3) *interact* with external contract last to give it smallest chance to exploit any unchanged context of current function for re-entering it

SOLIDITY – SECURITY PATTERNS

Checks-Effects-Interaction - example

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

contract ChecksEffectsInteractions {

    mapping (address => uint) balances;

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

    function withdraw(uint amount) public {
        //check
        require (balances[msg.sender] >= amount);
        //effect
        balances[msg.sender] -= amount;
        //interaction
        msg.sender.transfer(amount);
    }
}
```

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

contract ChecksEffectsInteractions {

    mapping (address => uint) balances;

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

    function withdraw(uint amount) public {
        //check
        require (balances[msg.sender] >= amount);
        //interaction
        (bool success,) = msg.sender.call{value: amount}("");
        require(success);
        //effect
        balances[msg.sender] -= amount;

    }
}
```

SOLIDITY – SECURITY PATTERNS

Checks-Effects-Interaction – safe interaction practice

Avoid *call* usage unless really necessary – i.e., for transferring currency just when it is known that this should specifically trigger some other action

Method	Forwarded gas	Exception handling
transfer	2300 (fixed)	throws on failure
send	2300 (fixed)	returns false
call{value: ...}	all gas (adjustable)	returns false

SOLIDITY – SECURITY PATTERNS

Mutex protection

- Purpose: additional protection besides Checks-Effects-Interaction towards re-entrancy attacks
- Implementation: usage of a mutex-style variable to prevent re-entrance in a contract function

```solidity
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

contract ChecksEffectsInteractions {

    mapping(address => uint) balances;
    bool mutex;

    modifier noReentrancy() {
        require(!mutex);
        mutex = true;
        _;
        mutex = false;
    }

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

    function withdraw(uint amount) public noReentrancy {
        require(balances[msg.sender] >= amount);
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

SOLIDITY – SECURITY PATTERNS

Emergency Stop

- Purpose: stop some critical contract functionality (e.g., funds withdrawal) in case a bug or other attack is detected as affecting the contract
- The pattern deals only with the *stop* part, not with the *detect* part
- Implementation:
  - Set a boolean stopping flag for critical functions
  - Protect critical functions via a modifier preliminary checking the activation of the stopping flag
  - Activate the flag in case of emergency via an access controlled transaction protected via an owner modifier

SOLIDITY – SECURITY PATTERNS

Emergency Stop - example

```
pragma solidity ^0.5.10;

contract EmergencyStop {

    bool isStopped = false;
    address payable owner;

    event DepositMade(uint);

    constructor () public {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender==owner);
        _;
    }

    modifier activeOperation {
        require(!isStopped);
        _;
    }

    function stopContract() public onlyOwner {
        isStopped = true;
    }

    function deposit() public payable activeOperation {
        emit DepositMade(msg.value) ;
    }
}
```

Various pattern variations can be implemented:

- Multiple stopping flags for different functionalities
- The stopped functionality can also be re-activated via a similar mechanism, resetting the stopping flag

SOLIDITY – SECURITY PATTERNS

Rate Limit

- Purpose: limit some critical contract functionality (e.g., funds withdrawal) to prevent any abusive repetition
- Implementation: typically relies on a time-based modifier

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

contract RateLimit {

    uint enabledAt = block.timestamp;

    modifier enabledEvery(uint t) {
        if (block.timestamp >= enabledAt) {
            enabledAt = block.timestamp + t;
            _;
        }
    }

    function criticalFunction() public enabledEvery(1 minutes) {
        // some code that should not be executed too often
    }
}
```

SOLIDITY – SECURITY PATTERNS

Balance Operation Limit

- Purpose: limit operations that use the *balance* field of the contract due to its potential unexpected changes

- Attack Situation:
  - Victim contract expects reaching a fixed amount balance in Ether (no extra sum) to permit funds withdrawal
  - Attacker contract with 1 Wei balance selfdestructs, which can transfer attacker's balance to victim contract without receive/fallback function call

- Implementation:
  - avoid using the *balance* field of the contract in critical value comparisons
  - keep the *balance* value in a different contract state variable and base the contract logic on that variable

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

The SWC Registry – a globally maintained list of weaknesses in smart contracts

- Hosted at: https://swcregistry.io/
- Some weaknesses are trivial
- Some examples are not very relevant or debatable
- Mapping to larger CWE (common software weaknesses) database

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC Registry

Smart Contract Weakness Classification and Test Cases

The following table contains an overview of the SWC registry. Each row consists of an SWC identifier (ID), weakness title, CWE parent and list of related code samples. The links in the ID and Test Cases columns link to the respective SWC definition. Links in the Relationships column link to the CWE Base or Class type.

ID	Title	Relationships	Test cases
SWC-136	Unencrypted Private Data On-Chain	CWE-767: Access to Critical Private Variable via Public Method	• odd_even.sol • odd_even_fixed.sol
SWC-135	Code With No Effects	CWE-1164: Irrelevant Code	• deposit_box.sol • deposit_box_fixed.sol • wallet.sol • wallet_fixed.sol
SWC-134	Message call with hardcoded gas amount	CWE-655: Improper Initialization	• hardcoded_gas_limits.sol
SWC-133	Hash Collisions With Multiple Variable Length Arguments	CWE-294: Authentication Bypass by Capture-replay	• access_control.sol • access_control_fixed_1.sol • access_control_fixed_2.sol
SWC-132	Unexpected Ether balance	CWE-667: Improper Locking	• Lockdrop.sol
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	• unused_state_variables.sol • unused_state_variables_fixed.sol • unused_variables.sol • unused_variables_fixed.sol

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

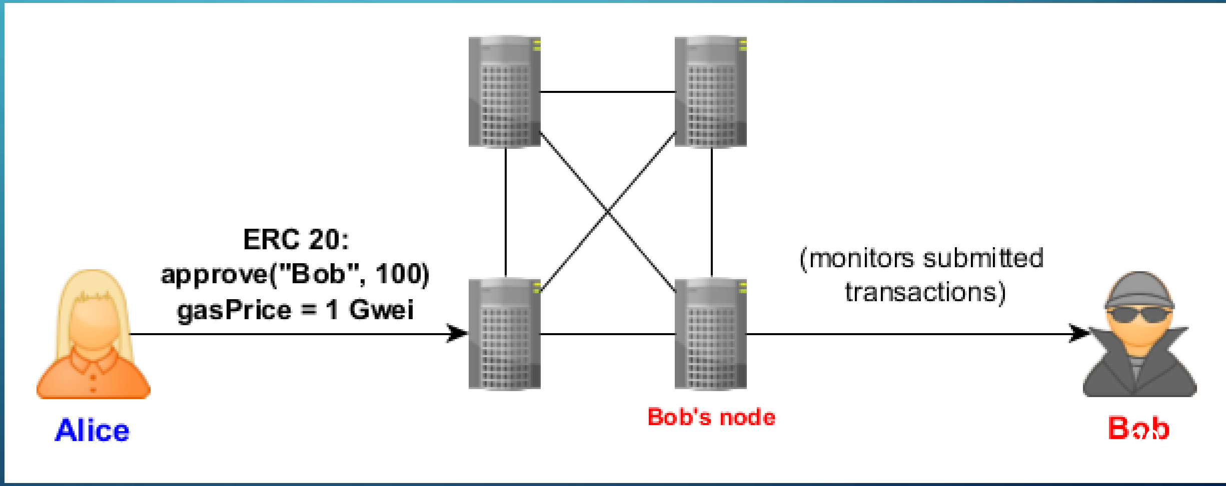
SWC 110 – Assert Violation

- Main idea:
  - use assert() only for invariant checks that *should never be false*
  - don’t use assert() for input validation checks, use require()
- Why?
  - failing assert() consumes all remaining gas
  - failing require() returns remaining gas
- More a „best practice” than a security weakness

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

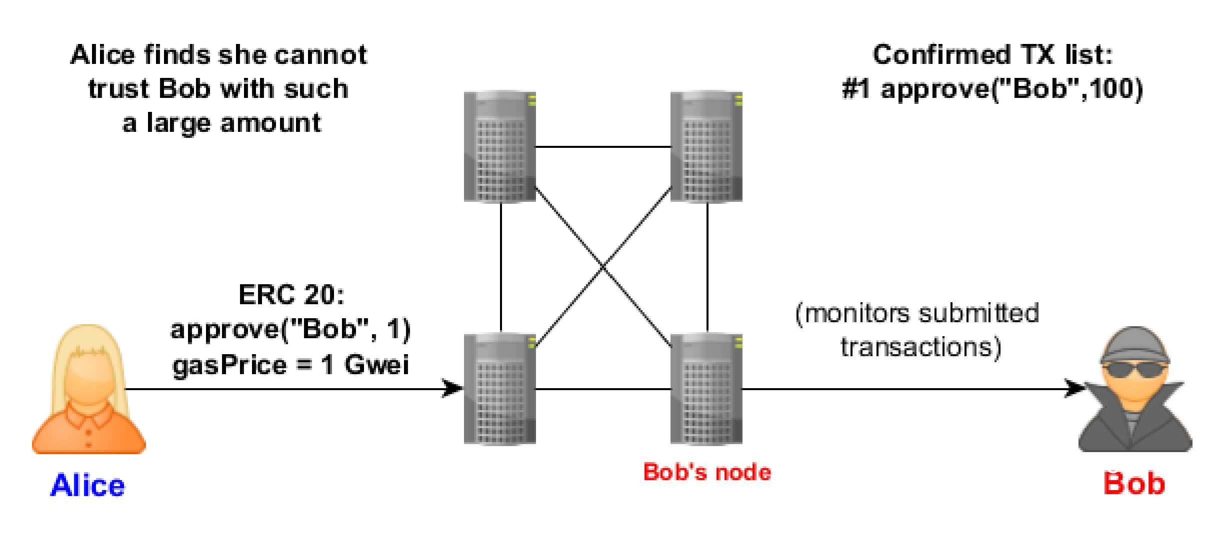
- Main idea:
  - A race condition might occur between submitted transactions, which permits exploit attempts
  - Main target: ERC 20 tokens



THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

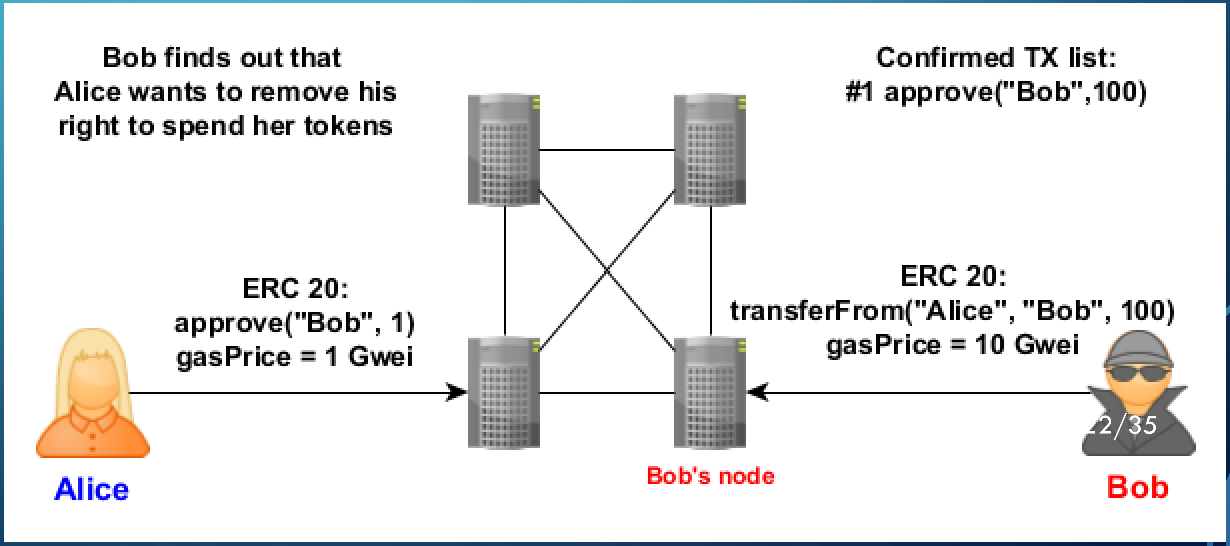
- Main idea:
  - A race condition might occur between submitted transactions, which permits exploit attempts
  - Main target: ERC 20 tokens



THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

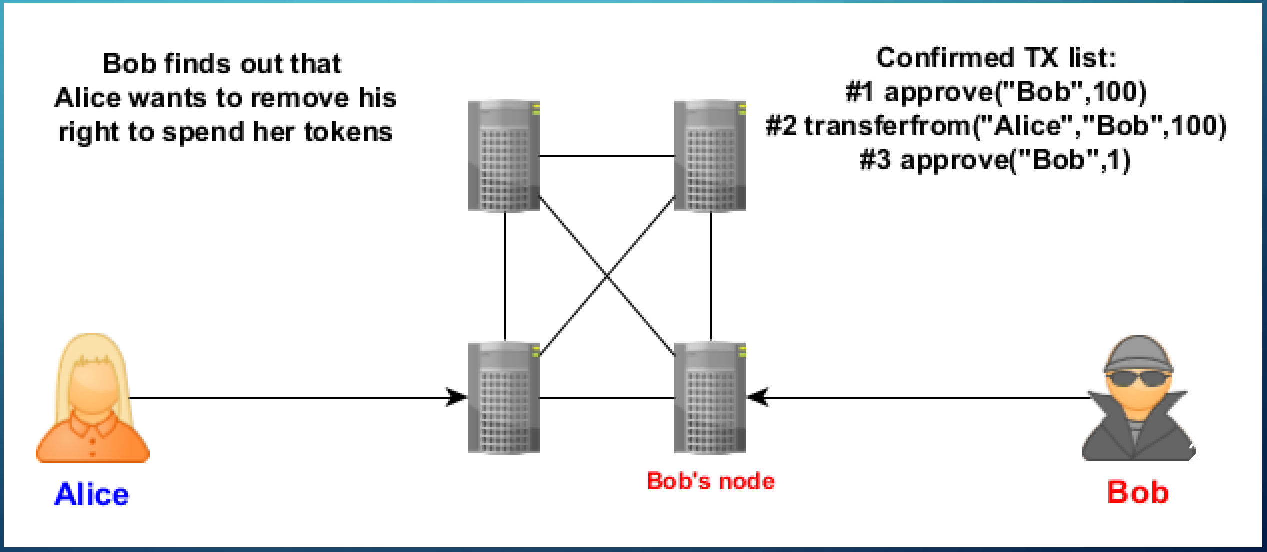
- Main idea:
  - A race condition might occur between submitted transactions, which permits exploit attempts
  - Main target: ERC 20 tokens



THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

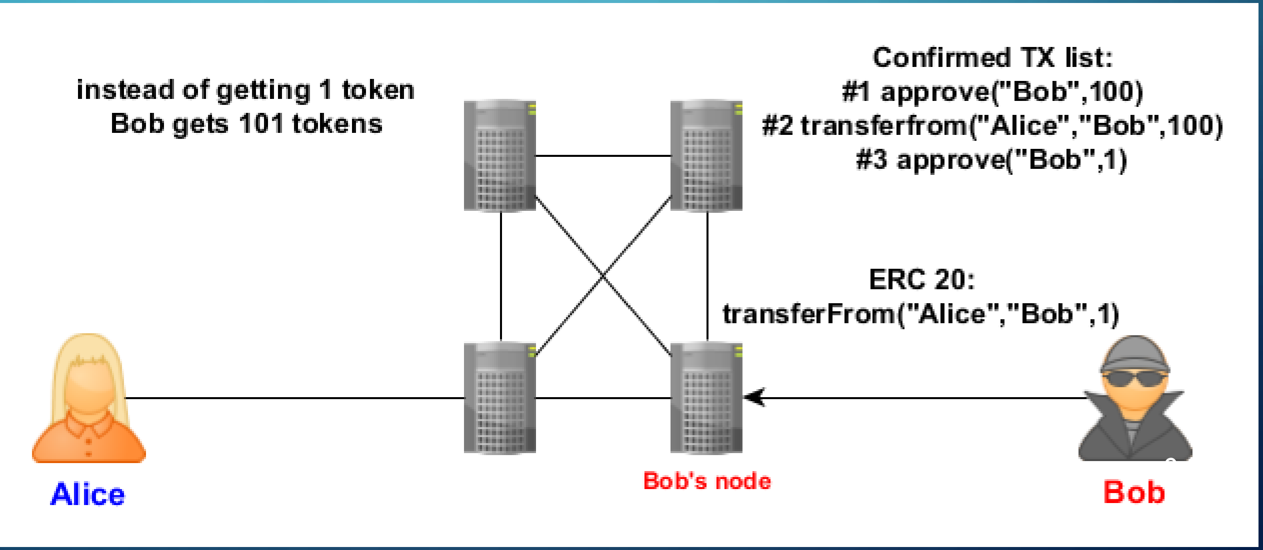
- Main idea:
  - A race condition might occur between submitted transactions, which permits exploit attempts
  - Main target: ERC 20 tokens



THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

- Main idea:
  - A race condition might occur between submitted transactions, which permits exploit attempts
  - Main target: ERC 20 tokens



THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

- Solution:
  - Add an extra parameter to the approve() method to indicate the expected allowance
    - Breaks ERC20 standard
    - Doesn't solve the 1st stealing attempt
  - ERC 20 official note:

NOTE: To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to `0` before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 114 – Transaction Order Dependence

- Attack can also target other use cases, e.g., getting rewards for solving a problem
- Solution: commit reveal hash pattern
  - #1 Alice sends A-hash = hash(salt, answer, address)
  - #2 Contract stores Alice address <-> A-hash
  - #3 Alice sends A-answer = (salt, answer)
  - #4 Bob sends B-answer = (salt, answer) with higher gasPrice
  - #5 Contract checks:
    - A-hash ≠ hash(B-answer, Bob address) -> denies reward to Bob
    - A-hash = hash(A-answer, Alice address) -> approves reward to Alice

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 115 – Authorization through tx.origin

- Main idea:
  - Do not use tx.origin for authorization, use msg.sender
- Why?
  - tx.origin identifies the first address initiating the transaction
  - msg.sender is the „last hop” in the transaction path
  - If authorization should be allowed also on calling contract basis, using tx.origin denies this (*apparently* more restrictive)
  - Malicious contracts can make use of a tx.origin received in a transaction to them to bypass an authorization in a victim contract

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

contract TxOriginVictim {

    address owner;
    constructor () {
        owner = msg.sender;
    }

    function transferTo(address to, uint amount) external {
        require (tx.origin == owner);
        (bool success,) = to.call{value: amount}("");
        require(success);
    }

    receive() payable external {}
}
```

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <=0.7.3;

interface TxOriginVictim {
    function transferTo(address to, uint amount) external;
}

contract TxOriginAttacker {
    address owner;

    constructor () {
        owner = msg.sender;
    }

    receive() payable external {
        TxOriginVictim(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Example adapted from: *Solidity – Tx Origin Attacks (Chris Coverdale – medium.com, 2018)*

- Exploit does not necessarily need to take place in a receive/fallback function (although the approach is more „stealthy")
- Any other function of the Attacker contract could call transferTo()

THE SWC REGISTRY – SMART CONTRACT WEAKNESSES CLASSIFICATION

SWC 120 – Weak Sources of Randomness from Chain Attributes

- Main idea:
  - Reliable randomness sources (seeds) are hard to obtain in smart contracts
  - don't use *block.timestamp* – a miner can alter that in a block
  - don't use *blockhash* – a miner can alter that in a block
  - don't use *block.difficulty* – a miner can alter that in a block
- Solution:
  - Best approach – use an oracle and an external source

SECURITY ANALYZERS FOR SMART CONTRACTS

Tools built for authomatic analysis of security:

- Some code sequences create vulnerabilities
- Generic patterns are associated to such code sequences
- Tools try to detect such patterns
- Level of detection implementation:
  - EVM bytecode
  - Smart contract code (Solidity, Vyper)
  - Other intermediate representation
- Security analyzers:
  - Mythril
  - Securify
  - NeuCheck
  - And others...

SECURITY ANALYZERS FOR SMART CONTRACTS

MYTHX

MythX – https://mythx.io/

- Security analysis platform using Mythril (developed by ConsenSys)
- Integrates with Remix and Truffle
- Free to use for its basic version
- MythX Pro – deeper vulnerability detection, charges a monthly subscription

SECURITY ANALYZERS FOR SMART CONTRACTS
SECURIFY

Securify:

- Started as an academic project at ETH Zurich
- Relies on detecting patterns in contract data flow graph:
    - compliance patterns (satisfaction of a security property)
    - violation patterns (negation of a security property)
- Classifies contract behavior according to detected patterns:
    - safe, unsafe and warnings
    - uses a semantic facts and inference rules formalization to describe and analyze bytecode patterns
- Used in smart contract auditing:
    - >18k contracts analyzed by the online tool (no more available currently)
    - >38 commercial audits

[reference: Securify – Practical Analysis of Smart Contracts, Tsankov et al., 2018]

SECURITY ANALYZERS FOR SMART CONTRACTS

SECURIFY – DETECTION PATTERN EXAMPLES

Stealing Ether pattern

- Similar to an attack on Parity wallet (~30M $ lost)
- *initWallet* function supposedly not known and called only in constructor
- 2-step attack: reset owner and withdraw funds
- Checked pattern: setting *owner* variable should depend on transaction initiator (msg.sender)

```
pragma solidity 0.5.10;

contract OwnableWallet {
    address payable owner;

    function initWallet(address payable _owner) {
        owner = _owner;
    }

    function withdraw(uint _amount) {
        if (msg.sender == owner) {
            owner.transfer(_amount);
        }
    }
}
```

(Example not currently compilable: access level required for functions)

SECURITY ANALYZERS FOR SMART CONTRACTS

SECURIFY – DETECTION PATTERN EXAMPLES

Frozen Funds pattern

- *walletLibrary* stores address of a public wallet library
- *deposit* can be used to send funds to contract
- *withdraw* delegates the call to the library to be executed in the current context
- If wallet library is disabled all wallet contract funds are blocked (happened in 2017: ~280M $)
- Checked pattern: ETH > 0 can be transferred *to* contract but contract does not provide any *call* to transfer ETH > 0 *from* contract

```


contract Wallet {
    address constant walletLibrary = ...;

    function deposit() payable {
        log(msg.sender, msg.value);
    }

    function withdraw() {
        walletLibrary.delegatecall(msg.data);
    }
}

```

DATA PRIVACY OVER BLOCKCHAIN

Blockchain platforms don't cope well with privacy *by design:*

- transparent – public transactions fully exposed
- lack of control over public nodes
- heavy crypto mechanisms computation is difficult

Alternative solutions: companion privacy layers

Probably most known platform: Secret Network/Enigma

https://scrt.network/

https://enigma.co/

DATA PRIVACY OVER BLOCKCHAIN

Enigma platform:

- started in 2015 and integrated by Secret Network in 2020

In the initial design:

- relies on *secure multi-party computation* (MPC) to provide data privacy
- basic idea:
    - data is split between nodes
    - a meaningful function is computed using multiple data pieces
    - no single party can access the complete data

Splitting data between nodes ≠ Data replication to nodes

- not using the blockchain structure (which is replicated)
- computation over private data is „off-loaded" to an external network layer
- a DHT accessible also through the blockchain is used to store references to off-loaded data

DATA PRIVACY OVER BLOCKCHAIN

Development model (initial design):

- designed to run on top of a layer 2 solution for Ethereum – the Enigma off-chain network
- *secret contracts* written in Rust, interoperable with Ethereum
- besides secure MPC based techniques the Enigma network relied also on TEEs (Trusted Execution Environments, i.e., Intel SGX) for data privacy preservation

Current state:

- rebranding during 2020 as Secret Network following a SEC ruling on securities law violation related to the ICO sale
- technology shift towards Cosmos/Tendermint
- more info at: https://build.scrt.network/dev/secret-contracts.html

DATA PRIVACY OVER BLOCKCHAIN

Quorum:

- Ethereum fork used in permissioned mode
- Initially developed for JP Morgan
- Acquired by Consensys in 2020
- Allows tagging transactions as private:
  - specific transaction parameter – *privateFor*
  - recipient public key used to encrypt transaction data
  - private transactions data can be accessed only by selected recipients