SOLIDITY SECURITY: BREAKING SMART CONTRACTS FOR FUN AND PROFIT

Mehdi Zerouali

sigma prime

$WHOAMI

- mehdi@sigmaprime.io – @ethzed
- Co-founder & Director @ Sigma Prime – https://sigmaprime.io
  - Blockchain & Cybersecurity Expertise
    - Distributed system design, niche & core blockchain components development
    - Offensive security assessments (pentests/red teaming)
    - Research in Blockchain space - Casper, Sharding, see https://github.com/sigp/lighthouse/
    - Smart contracts security reviews (cf. repo)
- Past: Penetration tester & Manager @ EY Advanced Security Centre
- Education: Telecom Engineering Masters @ INSA Lyon, France

sigma prime

AGENDA

- Ethereum Virtual Machine (EVM) 101
- Vulnerabilities, attack vectors & countermeasures
  - Default visibilities
  - Rounding issues
  - Arithmetic under/over flows
  - Re-entrancy
  - Unexpected ether
  - Entropy illusion
  - Race conditions & front running
  - *tx.Origin* for authentication
  - Denial of service
  - *Delegatecall*
- Road ahead / ETHSecurity experience feedback

ETHEREUM VIRTUAL MACHINE 101

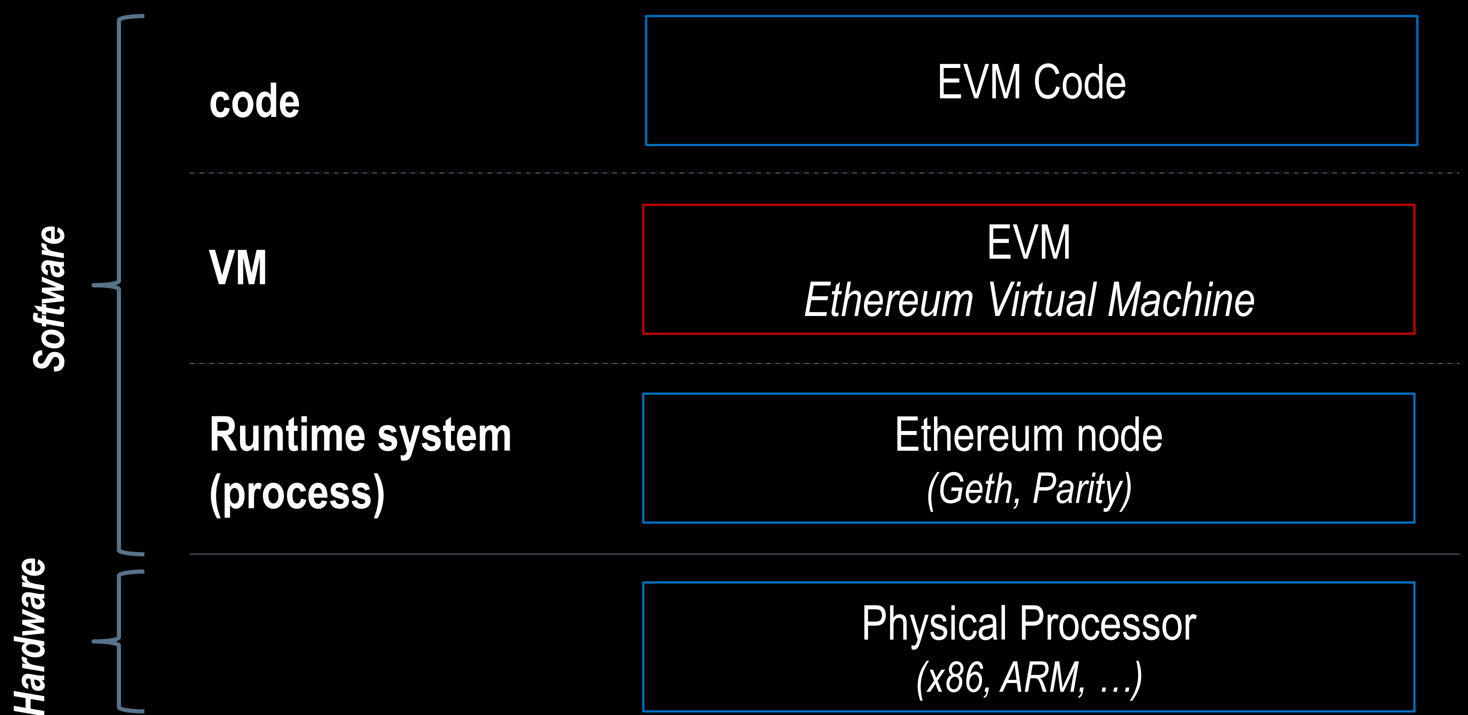
```
contract Rot13Encryption {

  event Result(string convertedString);

    //rot13 encrypt a string
    function rot13Encrypt (string text) public {
        uint256 length = bytes(text).length;
        for (var i = 0; i < length; i++) {
            byte char = bytes(text)[i];
            //inline assembly to modify the string
            assembly {
                char := byte(0,char) // get the first byte
                if and(gt(char,0x6D), lt(char,0x7B)) // if the character is in
                { char:= sub(0x60, sub(0x7A,char)) } // subtract from the ascii
                if iszero(eq(char, 0x20)) // ignore spaces
                {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))} // add 1
            }
        }
        Result(text);
    }
```

EVM 101

- Sandboxed Virtual Stack Machine
  - Embedded within all (full) Ethereum nodes
  - Responsible for executing contract bytecode
  - Runtime environment



sigma prime

EVM 101

- Smart contracts are made of bytecodes stored at particular addresses

- Smart contracts are typically written in higher level languages
  - e.g. Vyper, Solidity, LLL, Bamboo
  - *Note: Solidity can include inline assembly! Use at your own risks!*

- These higher level languages then compile into EVM bytecode

EVM 101

```solidity
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}

```

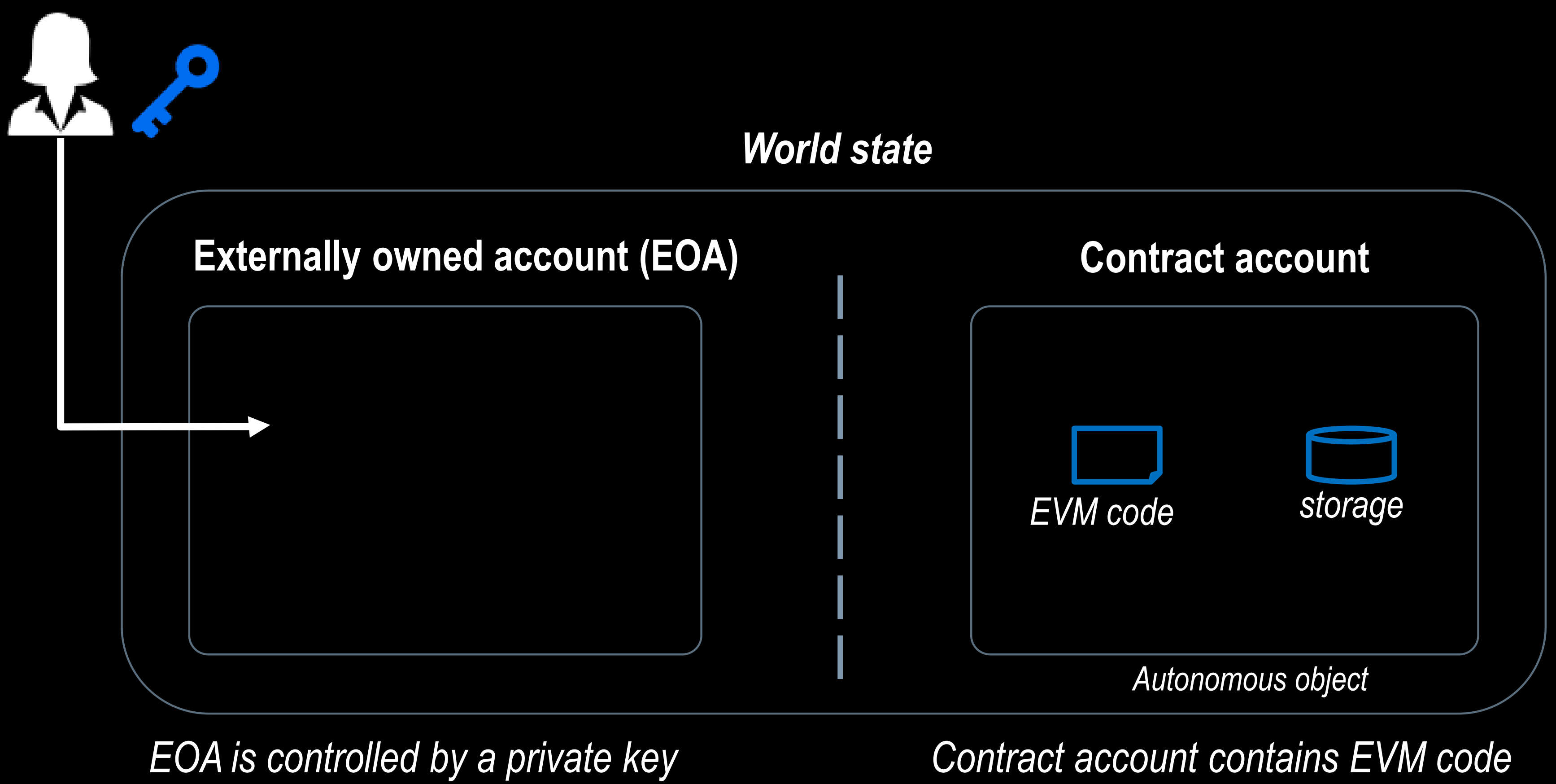
```
⚙ mz@ZedSigP  ~/Ops  solc --bin-runtime SimpleStorage.sol

======= SimpleStorage.sol:SimpleStorage =======
Binary of the runtime part:
6080604052600436106049576000357c01000000000000000000000000000000000000
00000000000000000000900463ffffffff16806360fe47b114604e5780636d4ce63c14
6078575b600080fd5b348015605957600080fd5b506076600480360381019080803590
6020019092919050505060a0565b005b348015608357600080fd5b50608a60aa565b60
40518082815260200191505060405180910390f35b8060008190555050565b60008054
9050905600a165627a7a723058203e4a4307a3717359af8d9e394e5bdce06379ec6d97
e126511e388d4144a42e960029
⚙ mz@ZedSigP  ~/Ops
```

sigma prime

EVM 101

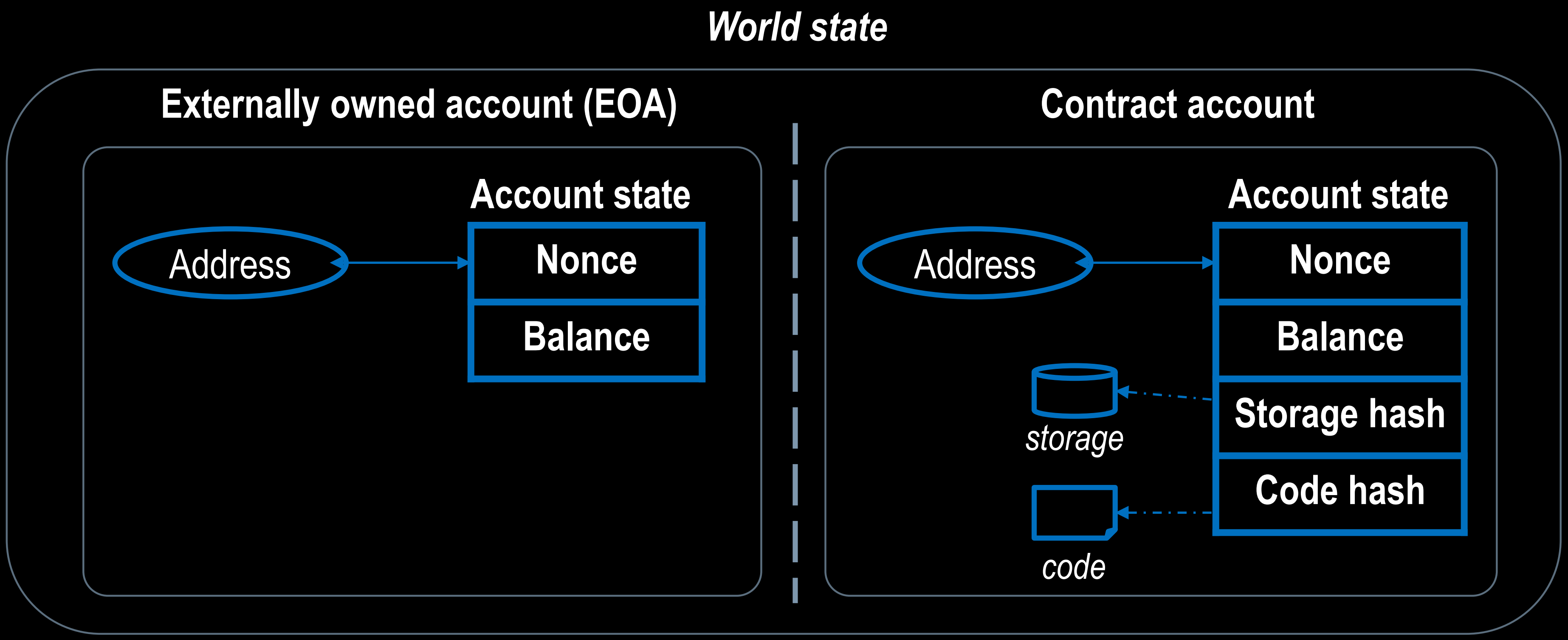
- Distinction between externally owned accounts and contracts



sigma prime

EVM 101

- Distinction between externally owned accounts and contracts



EOA is controlled by a private key
EOA cannot contain EVM code

Contract is controlled by EVM code
Contract contains EVM code

EVM 101

- EVM uses "pseudo"-registers – not standard registers like other VMs
- Defines a set of Opcodes - cf Gav's yellow paper, including:
  - Arithmetic operations (ADD, SUB, MUL, DIV, …)
  - Bitwise operations (AND, OR, XOR,…)
  - Context & block information (GASPRICE, GASLIMIT, NUMBER, …)
  - Stack, Memory & Storage operations (PUSH, POP, MSTORE, SSTORE, MLOAD, SLOAD)
  - Cryptographic function (SHA3)
  - …

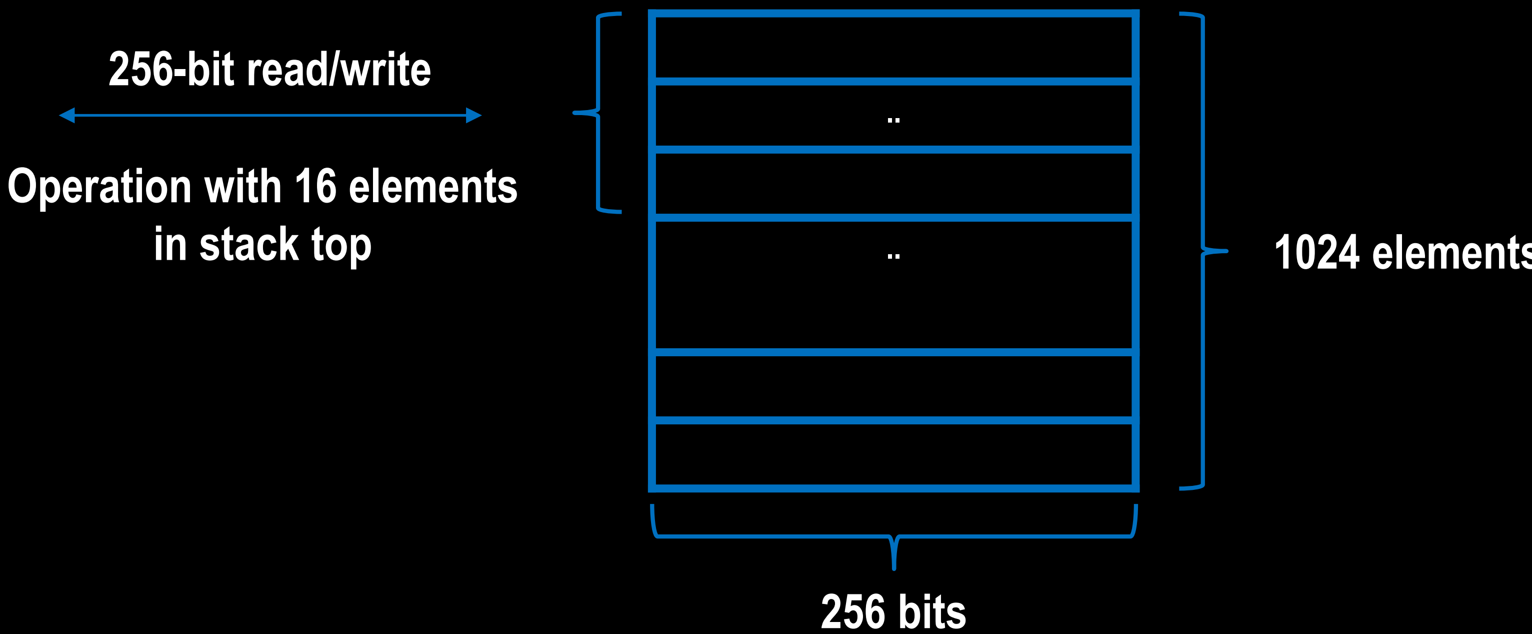
sigma prime

EVM 101

- No registers = all instructions invocation (and parameter passing) are performed via the EVM stack
- EVM uses 160-bit addresses
- EVM outputs logs
- EVM introduces the concept of Gas
- Important distinction: Stack, Memory & Storage

EVM 101

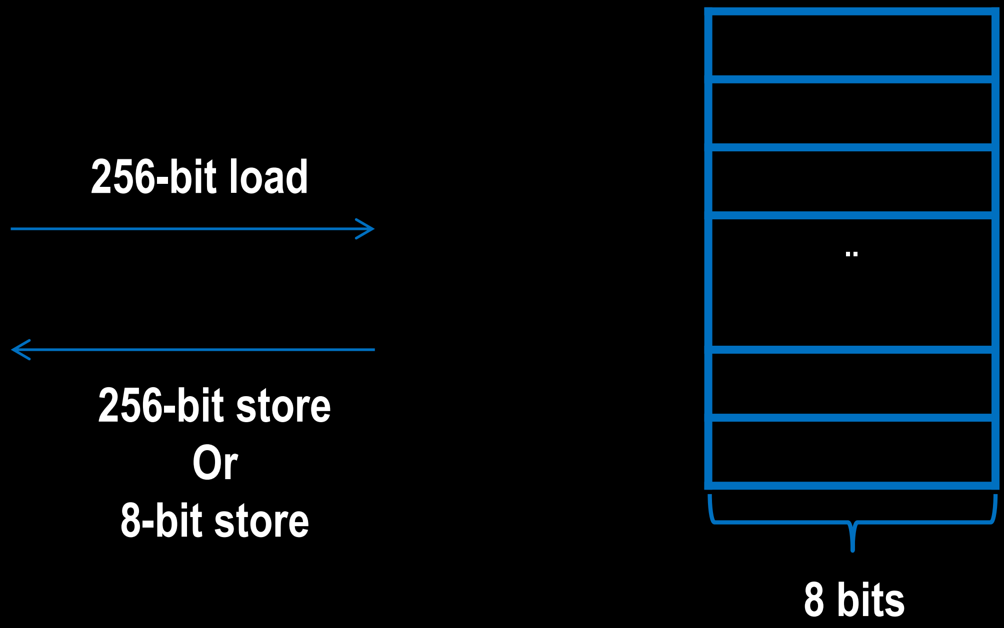
- Stack:



- All EVM operations are performed on the stack
- Accessed via PUSH/POP/COPY/SWAP/etc

EVM 101

- Memory:

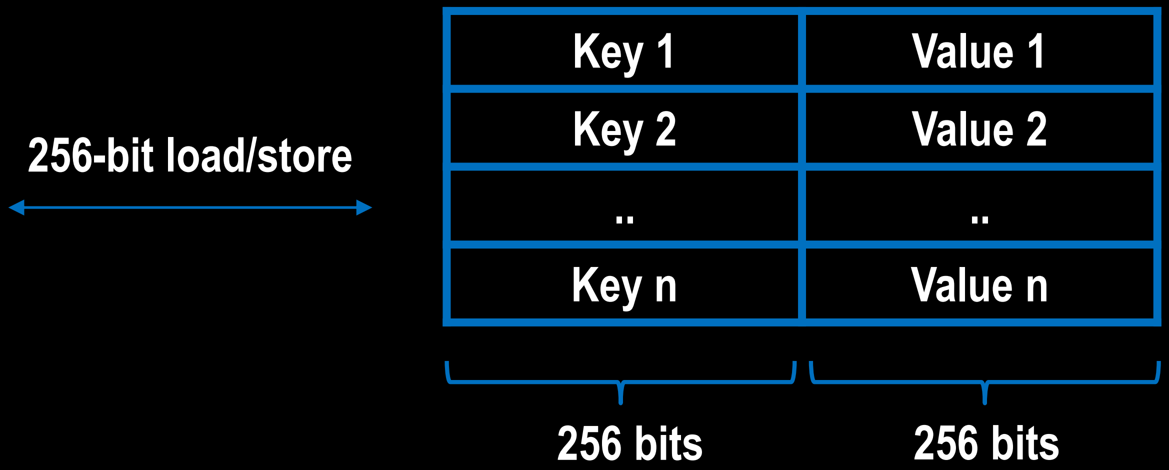


- Volatile memory, refreshed and cleared for each message call
- Accessed via MSTORE/MLOAD
- Memory is more costly the larger it grows (scales quadratically)

sigma prime

EVM 101

- Account storage:



- Persistent memory area, declared outside of user-defined functions
- Accessed via SSTORE/SLOAD
- Costly to read and very expensive to write

EVM 101

- Contract Application Binary Interface (ABI): standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction
  - Data is encoded according to its type
  - Encoding is not self describing and thus requires a schema in order to decode

```
⚙ mz@ZedSigP  ~/Ops  solc --abi SimpleStorage.sol

======= SimpleStorage.sol:SimpleStorage =======
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"}]
⚙ mz@ZedSigP  ~/Ops 
```

EVM 101

- Function signature = `function_name(arg_type1,arg_type2)`
  - e.g. `transfer(uint256,uint256)`
- Function selectors:
  - 4 bytes of the Keccak (SHA-3) hash of the signature of the function
  - Allows calling specific functions in the contract
    - e.g. `keccak(transfer(uint256,uint256))`

sigma prime

SOLIDITY PITFALLS

```
contract Rot13Encryption {

  event Result(string convertedString);

   //rot13 encrypt a string
   function rot13Encrypt (string text) public {
      uint256 length = bytes(text).length;
      for (var i = 0; i < length; i++) {
         byte char = bytes(text)[i];
         //inline assembly to modify the string
         assembly {
            char := byte(0,char) // get the first byte
            if and(gt(char,0x6D), lt(char,0x7B)) // if the character is in
            { char:= sub(0x60, sub(0x7A,char)) } // subtract from the ascii
            if iszero(eq(char, 0x20)) // ignore spaces
            {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))} // add 1
         }
      }
      Result(text);
   }
```

sigma prime

DEFAULT VISIBILITIES (1/3)

- Consider this trivial contract that acts like an address guessing bounty game
- To win the balance of the contract, a user must generate an Ethereum address whose last 8 hex characters are 0

```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0.
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

DEFAULT VISIBILITIES (2/3)

- Problem: visibility not specified!
- Defaults to `public`
- Any address can call `_sendWinnings()` to steal the bounty

- Mitigation:
  - Always specify the visibility of all functions in a contract, even if they are intentionally public
  - Solc will throw warnings that have no explicit visibility set

DEFAULT VISIBILITIES (3/3)

- Real-World Example: Parity MultiSig Wallet (1st Hack)

```
contract WalletLibrary is WalletEvents {

  // constructor is given number of sigs required to do protected "onlymanyowners" transactions
  // as well as the selection of addresses capable of confirming them.
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }

  ...

  // constructor - just pass on the owner array to the multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit) {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
}
```

FLOATING POINTS AND PRECISION (1/3)

- Problem: There is no fixed point type in Solidity (yet), developers are required to implement their own using the standard integer data types

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() public payable {
        uint tokens = msg.value/weiPerEth*tokensPerEth;
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth); //
    }
}
```

sigma prime

FLOATING POINTS AND PRECISION (2/3)

- Mathematical calculations for buying and selling tokens are correct but lack of floating point numbers will give erroneous results:
  - Buying: If the value is less than 1 ETH the initial division will result in 0, leaving the final multiplication 0
  - Selling: Less than 10 tokens will result in 0 ETH

FLOATING POINTS AND PRECISION (3/3)

- Mitigations:
  - Ensure that any ratios or rates you are using allow for large numerators in fractions:
    - Use `weiPerTokens` instead of `tokensPerEth`
  - Keep in mind the order of operations:
    - `msg.value*tokenPerEth/weiPerEth` instead of `msg.value/weiPerEth*tokenPerEth`
  - Use safe libraries that allow for floating points (Cf MakerDAO’s DSMath)

sigma prime

ARITHMETIC UNDER/OVER FLOWS (1/4)

- An integer variable, only has a certain range of numbers it can represent.
  - uint8 can only store numbers in the range [0,2^8-1] = [0,255]
  - Trying to store 256 will result in 0 (and 257 in 1)
- For uint256, the range is [0,2^256-1]
- Similar to adding 2π to the angle of a trigonometric function:
  - sin(x)=sin(x+2π)
  - cos(x)=cos(x+2π)

sigma prime

ARITHMETIC UNDER/OVER FLOWS (2/4)

```
pragma solidity ^0.4.18;

contract Token {

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

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public constant returns (uint balance)
    return balances[_owner];
  }
}
```

sigma prime

ARITHMETIC UNDER/OVER FLOWS (3/4)

- Mitigation: use mathematical libraries which replace the standard math operators (additions, subtractions & multiplications)
  - Most common one is OpenZeppelin’s SafeMath
  - We’ve seen clients and projects purposefully avoid this to save gas… Be careful!

sigma prime

ARITHMETIC UNDER/OVER FLOWS (4/4)

- Real-world examples:
  - 4chan group's ponzi scheme PoWHC (866 ETH stolen)
  - batchTransfer bug (ERC20) = integer underflow

```
  function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i], _value);
    }
    return true;
  }
}
```

UNEXPECTED ETHER (1/5)

- When ETH is sent to a contract, it must execute:
  - Either the fallback function
  - Or another function described in the contract
- 2 exceptions (where ETH can be sent to a contract without executing code)
  - Self-destruct (suicide)
    - Any contract can implement `selfdestruct(target)`, where all funds are transferred to *target* after all bytecode from the contract is removed
  - Pre-sent ETH
    - Contract addresses are deterministic:
    - Address = `sha3(rlp.encode(account_address, tx_nonce))`

sigma prime

UNEXPECTED ETHER (2/5)

- Lots of smart contracts rely on invariant-checking as a defensive programming technique
  - Good design, providing the variable(s) checked is actually invariant (e.g. ERC20 total supply)
- Current ETH stored in the contract (i.e. this.balance) is not an invariant!

sigma prime

UNEXPECTED ETHER (3/5)

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }
```

sigma prime

UNEXPECTED ETHER (4/5)

```
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

UNEXPECTED ETHER (5/5)

- Mitigation techniques:
  - Contract logic should avoid being dependent on exact values of the contract balance since it can be artificially manipulated
  - If exact values of deposited ETH are required, define a dedicated variable to this purpose

ENTROPY ILLUSION (1/3)

- All transactions on the Ethereum blockchain are deterministic state transition operations
- There is no `rand()` function in Solidity
- Achieving decentralised entropy (randomness) is a well established problem
  - See RandDAO’s by VB for using a chain of hashes
- Some of the first contracts were based around gambling
  - Common pitfall:
    - Using block variables (hashes, timestamps, etc.)
      - Controlled by the miners!
    - Thinking that a *seed* is private
      - A `private` variable can still be read!

sigma prime

ENTROPY ILLUSION (2/3)

- Mitigation: Commit-reveal approach:
  - A *commit* stage, when parties submit their cryptographically protected secrets to the smart contract
  - A *reveal* stage, when parties announce cleartext seeds, the smart contract verifies that they are correct, and the seeds are used to generate a random number

- Exciting future: Verifiably Delayable Functions (VDF)
  - Included in Ethereum 2.0 and used for the Beacon Chain

ENTROPY ILLUSION (3/3)

- Real-world examples:
  - See https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620
  - 3649 live smart contracts analysed with some sort of pseudo random number generator (PRNG)
  - Found 43 contracts which could be exploited
  - Discusses the pitfalls of using block variables for entropy

FRONT-RUNNING / RACE CONDITIONS (1/3)

- Consider this simple hash-guessing contract:

```
contract FindThisHash {
    bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f59511

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

FRONT-RUNNING / RACE CONDITIONS (2/3)

- Attackers can watch the transaction pool for anyone submitting a solution
- They can verify it's validity (solution here is *Ethereum!*) and submit an equivalent transaction with a much higher **`gasPrice`** than the original
- 2 main types of front-running:
  - User front running (significantly worse) – see example
  - Miners front running
    - Miners can order tx however they feel like

FRONT-RUNNING / RACE CONDITIONS (3/3)

- Mitigation:
  - Create upper bounds on the `gasPrice` (doesn’t fix miner front-running)
  - Use commit-reveal pattern (see ENS)

- Real-world examples:
  - Bancor: Front running
  - ERC-20: Front running in `approve` method

TX.ORIGIN AUTHENTICATION (1/3)

- Solidity has a global variable – **tx.origin** – which traverses the entire call stack and returns the address of the account that originally sent the call/transaction
- Using this variable for authentication in Smart Contracts leaves the contract vulnerable to phishing attacks

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

TX.ORIGIN AUTHENTICATION (2/3)

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attackers address to receive funds.

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () {
        phishableContract.withdrawAll(attacker);
    }
}
```

- Attacker can publish contract above, and convince the owner to send this contract some ETH!

TX.ORIGIN AUTHENTICATION (3/3)

- Mitigation:
  - Do not use **`tx.origin`** for authentication/authorisation in smart contracts
  - If we want to deny external contracts from calling a contract:
    - **`require(tx.origin == msg.sender)`**

sigma prime

DENIAL OF SERVICE (1/3)

- Looping through externally manipulated mappings or arrays:

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of tokens
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

DENIAL OF SERVICE (2/3)

- Owner operations:

```
bool public isFinalized = false;
address public owner; // gets set somewhere

function finalize() public {
    require(msg.sender == owner);
    isFinalized == true;
}

// ... extra ICO functionality

// overloaded transfer function
function transfer(address _to, uint _value) returns (bool) {
    require(isFinalized);
    super.transfer(_to,_value)
}
```

sigma prime

DENIAL OF SERVICE (3/3)

- Mitigations:
  - Do not loop through data structures that can be artificially manipulated by external users!
    - Withdrawal pattern recommended
    - In-line Assembly allows to return the last *transfer* processed for batch transfers
  - Use a fail-safe in case owner becomes incapacitated (set up the owner as a multi-sig wallet)
  - Use a timelock:
    - `require(msg.sender == owner || now > unlockTime)`

sigma prime

RE-ENTRANCY (1/4)

- Contracts can call other contracts
  - Referred to external calls
- Fallback function:
  - Unnamed function, external visibility
  - Executed on a call to a contract if no other function called
- External calls can be hijacked by attackers to execute further code, including calls back into itself
- Attacker can carefully construct a contract at an external address which contains malicious code in the fallback function
- When a contract sends ETH to this address, it will invoke the malicious code
- Typically the malicious code executes a function on the vulnerable contract, performing operations not expected by the developer
- Called re-entrancy because malicious calling contract “re-enters” code execution

RE-ENTRANCY (2/4)

```
contract EtherStore {

    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

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

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

RE-ENTRANCY (3/4)

```
import "EtherStore.sol";

contract Attack {
  EtherStore public etherStore;

  // intialise the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function pwnEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // fallback function - where the magic happens
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

RE-ENTRANCY (4/4)

- Mitigation strategies:
  - Use built-in transfer() or send() instead
    - Limited to 2,300 gas => can only log an event, can’t reenter
  - Make sure all state changing code happens BEFORE ETH is sent out (or any external call)
  - Introduce a mutex – state variable which locks the contract during code execution
- Real-world examples:
  - The DAO!

DELEGATE CALL (1/8)

- **CALL** and **DELEGATECALL** opcodes allow devs to modularise their code
- **DELEGATECALL** allows the use of libraries – reusable code for future contracts
- **DELEGATECALL** preserves contract context:
  - Code executed via **DELEGATECALL** will act on the **<u>storage of the calling contract</u>**
- To understand the issue, let’s take a look at storage/state variables
  - Storage variables get put into *slots*
  - See previous slides and Solidity docs

DELEGATE CALL (2/8)

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}

```

- storedData => slot[0]
  - storedData2 => slot[1]; storedData3 => slot[2]

DELEGATE CALL (3/8)

ownerLib => slot[0]
simpleUserLib => slot[1]

```
pragma solidity ^0.4.18;

contract DelegateCallee {
  //acts as a Library
  address ownerLib;
  address simpleUserLib;

  function changeUserLib(address _newUser) public {
    simpleUserLib = _newUser;
  }

  function changeOwnerLib(address _newOwner) public {
    ownerLib = _newOwner;
  }
}
```

sigma prime

DELEGATE CALL (4/8)

```
simpleUser => slot[0]
owner => slot[1]
delegateCallLibrary => slot[2]
```

```
pragma solidity ^0.4.18;

contract DelegateCaller {
  address simpleUser;
  address owner;
  address delegateCallLibrary;
  bytes4 constant changeUserSig = bytes4(sha3("changeUserLib(address)"));
  bytes4 constant changeOwnerSig = bytes4(sha3("changeOwnerLib(address)"));

  constructor(address _delegateCallLibrary) public {
    delegateCallLibrary = _delegateCallLibrary;
  }

  function setUser(address _newUser) public {
    delegateCallLibrary.delegatecall(changeUserSig, _newOwner));
  }

  function setOwner(address _newOwner) public {
    delegateCallLibrary.delegatecall(changeOwnerSig, _newOwner));
  }

  function() public {
    delegateCallLibrary.delegatecall(msg.data);
  }
}
```

sigma prime

DELEGATE CALL (5/8)

- The use of **DELEGATECALL** can lead to unexpected code execution
- Because the addressing layout is not the same (owner in slot[0] for Library and in slot[1] for caller):
  - Calling **setUser** will actually change the owner
  - Calling **setOwner** will actually change the simpleUser
- Important to remember that the state changing is the caller’s
- **DELEGATECALL** is often used for smart contract upgrades

sigma prime

DELEGATE CALL (6/8)

- Real-world example: Parity MultiSig Wallet Hack #2
  - “I accidentally killed it” *devop199*

```
contract WalletLibrary is WalletEvents {
  ...
  // throw unless the contract is not yet initialized.
  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

  // constructor - just pass on the owner array to the multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit)
  only_uninitialized {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }

  // kills the contract sending everything to `_to`.
  function kill(address _to) onlymanyowners(sha3(msg.data)) external {
    suicide(_to);
  }
  ...
}
```

DELEGATE CALL (7/8)

- Wallet contract essentially passes all calls to the WalletLibrary contract via a delegate call.

```
contract Wallet is WalletEvents {
  ...
  // METHODS
  // gets called when no other function matches
  function() payable {
    // just being sent some cash?
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }
  ...
  // FIELDS
  address constant _walletLibrary = 0xcafecafecafecafecafecafecafecafecafecafe
}
```

sigma prime

DELEGATE CALL (8/8)

- But the “Library” has its own state! It’s actually a contract!
- Devops199 first called **`initWallet()`**
  - Obtained ownership of the library
- Then called **`kill()`**
  - Modifier passes
  - Code associated with this contract is deleted
  - All wallets that were referencing the library become unusable
  - $150M+ stuck

IS THAT IT?

- A lot more vulns, can't cover them all!
  - External contract referencing
  - Short address attacks
  - Unchecked CALL return values
  - Uninitialised storage pointers
  - Constructors issues
- We collected all known solidity vulns here:
  - https://blog.sigmaprime.io/solidity-security.html
- Included in "Mastering Ethereum" by Andreas Antonopoulos
- It's open source – feel free to contribute:
  - https://github.com/sigp/solidity-security-blog

ROAD AHEAD

ETHSECURITY UNCONF FEEDBACK

sigma prime

ETHSECURITY UNCONFERENCE 0

- @Berlin on Sept 6th
- ~100 people
  - SC auditors, SC developers, security tool developers, non-technical people
  - Orgs represented: Parity, Trail of Bits, ConsenSys Diligence, Sigma Prime, Ethereum Foundation, etc.
- First formal informal gathering
  - Next step: get together @ Devcon4
- Different objectives depending on participants
- First focus on smart contract security
  - Scope will most likely be extended later

ETHSECURITY UNCONFERENCE 0

- Topics covered:
  - Standards/Guidelines
  - Assessment stamps
  - Open source code assessments
  - Smart contract upgradeability
  - Governance structure
    - Avoid OWASP-model failure
    - Engineers vs Salesmen (prevent vendor shilling)
  - Funding
    - 1x full-time job

sigma prime

QUESTIONS?

mehdi@sigmaprime.io

@ethzed

sigma prime